

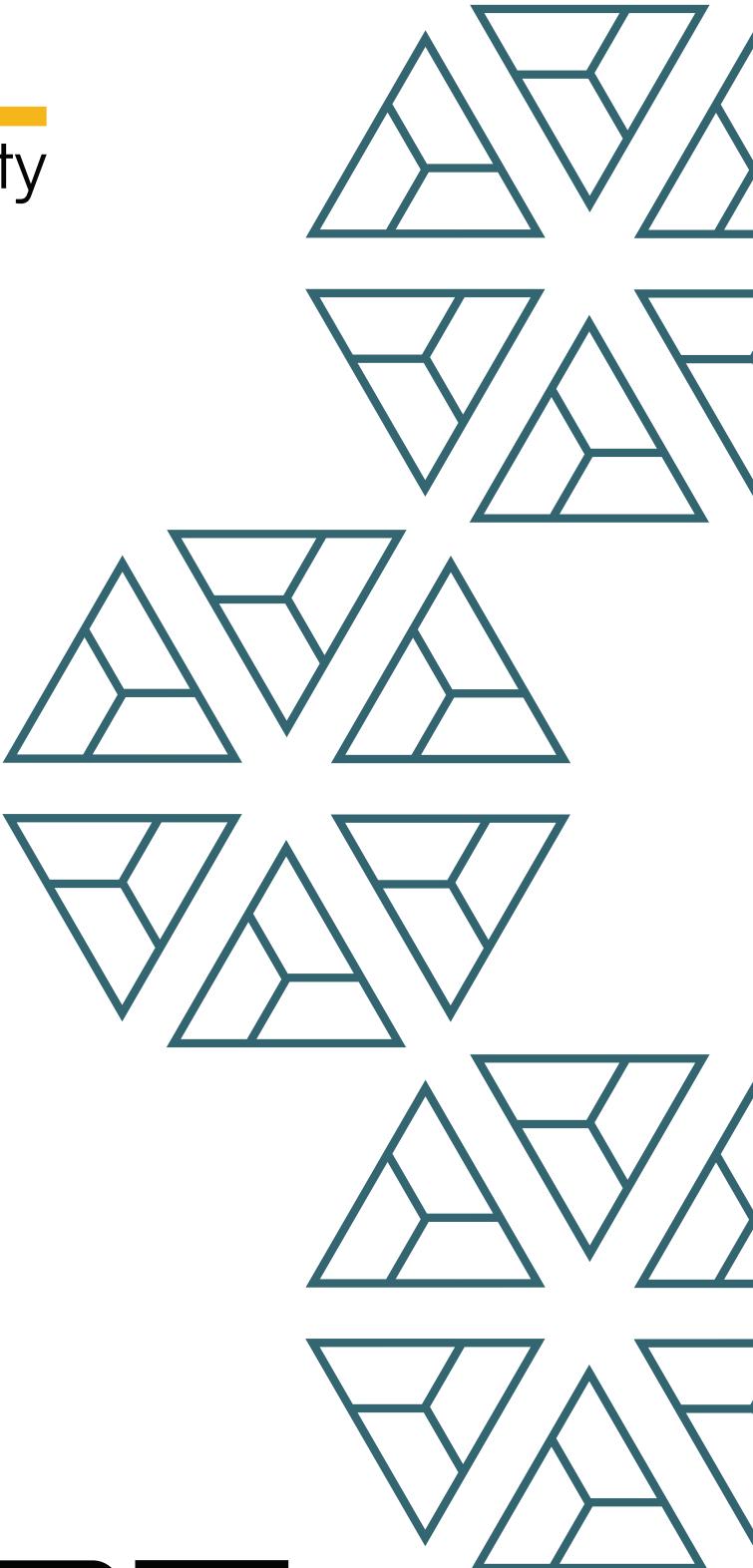


**BALL**  
security

RocketPool  
Saturn

# **FINAL REPORT**

January '2026



## Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

## 1. Project Details

### Important:

Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	RocketPool - Saturn
Website	rocketpool.net
Language	Solidity
Methods	Manual Analysis
Github repository	<a href="https://github.com/rocketpool/rocketpool/tree/02621fa4f88af44b7e0475840b222b9aa ba4a6dd/contracts/contract">https://github.com/rocketpool/rocketpool/tree/02621fa4f88af44b7e0475840b222b9aa ba4a6dd/contracts/contract</a>
Resolution 1	<p><a href="https://github.com/rocketpool/rocketpool/tree/de2383079bb617f5f6eaa5691ae6ae5c769754d4">https://github.com/rocketpool/rocketpool/tree/de2383079bb617f5f6eaa5691ae6ae5c769754d4</a></p> <p><b>Note:</b> Multiple contracts/functionalities have been refactored to certain degrees. These refactorings cannot be covered under a standard resolution round. The following contracts/specific control-flows that touch these contracts do fall under this:</p> <ul style="list-style-type: none"><li>- <b>RocketDAOProtocolSettingsMegapool:</b> This contract has multiple new variables introduced. It must be reviewed at the same time the new Megapool logic is reviewed.</li><li>- <b>DepositPool:</b> This contract has changes regarding the crediting logic and it is severely tied to the NodeDeposit and Megapool contracts and plays a core role in the crediting mechanism. It must be reviewed at the same time when the crediting mechanism and all related control flows are being reviewed. Other than that, no parts of this contract require a re-review</li></ul>

- **RocketMegapoolDelegate:** The contract experienced significant refactoring combined with a lot of identified issues during the initial audit round this will require a full re-review.
- **RocketMegapoolManager:** This contract has a lot of changes and is directly involved in Megapool creation and Megapool transitions. It is required to fully review this contract in the context of Megapool creations and transitions. Furthermore, the proof mechanism was refactored.
- **RocketMegapoolPenalties:** The changes which have been applied are mainly related to using `block.timestamp` instead of `block.number`, including a new contract “`RocketNetworkSnapshotsTime`”. There are no obvious issues but we would recommend a careful check including the new `RocketNetworkSnapshotsTime` contract
- **RocketMegapoolStorageLayout:** Inherently included due to MegapoolDelegate changes
- **RocketNetworkPenalties:** The contract has been refactored to use `block.timestamp` instead of `block.number` with the `RocketNetworkSnapshotsTime` contract. There are no obvious issues with this mechanism and since the running totals was never a live-implementation, there will be no historical inconsistencies. However, we still recommend reviewing this change fully.
- **RocketNetworkRevenues:** This contract has been refactored to use a time-based version of the snapshot mechanism and capital ratio related to Megapools. This new mechanism needs to be fully reviewed - especially the capital ratio introduction in incorporation with

different Megapool transitions.

- **RocketNodeDeposit:** This contract plays a critical role in the crediting mechanism and the Megapool creation. Since there were many issues to the crediting mechanism that required fixes in different spots, it is required to fully review the new credit logic, including implication on the network balance. It has to be noted that there are only small changes to this contract. No full review of this contract is required, it is only used to illustrate the control-flow for validator additions with credits.
- **RocketMerkleDistributorMainnet:** The claim control-flow has been refactored to support old and new claims in the same function. This refactoring must be fully audited to make sure no new issues have been introduced.
- **RocketRewardsPool:** The control-flow of `_executeRewardSnapshot` with the `treeVersion` parameter must be validated. The contract itself does not require a re-audit.
- **RocketUpgradeOneDotFour:** The contract has been refactored and there are new instances which need to be accounted for such as the `RocketNetworkSnapshotTime` contract and new variables that need to be initialized
- **BeaconStateVerifier:** A new slot-proof mechanism has been implemented which is tied to refactoring of Megapool logic. This must be audited isolated in itself for correctness as well as the corresponding usage within Megapools

Fixes for these contracts will remain in “open” status with corresponding comments.

Resolution 2/Follow-Up	<p><a href="https://github.com/rocketpool/rocketpool/tree/c9d9cbf2288ccbffe014a442b0ee458dcfc3dfe1">https://github.com/rocketpool/rocketpool/tree/c9d9cbf2288ccbffe014a442b0ee458dcfc3dfe1</a></p>
	<p>The following contracts were reviewed within the scope of contextual changes:</p> <ul style="list-style-type: none"><li>- <b>RocketDAOProtocolSettingsMegapool:</b> <a href="https://www.diffchecker.com/lGKpwJE8/">https://www.diffchecker.com/lGKpwJE8/</a><ul style="list-style-type: none"><li>- change notifyThreshold to epochs</li><li>- implement user.distribute.delay/user.distribute.delay.shortfall</li><li>- implement megapool.penalty.threshold</li><li>- implement megapool.dissolve.penalty</li></ul></li><li>- <b>DepositPool:</b> <a href="https://www.diffchecker.com/cWhwiBV0/">https://www.diffchecker.com/cWhwiBV0/</a><ul style="list-style-type: none"><li>- handling of proper nodeBalanceKey increase during nodeDeposit</li><li>- nodeBalanceKey decreased during exitQueue (dequeue control-flow)</li><li>- applyCredit does not increase nodeBalanceKey anymore</li><li>- withdrawCreditFor functionality</li><li>- ensure debt is zero before credit is withdrawn</li><li>- nodeBalanceKey not decreased during credit withdrawal</li><li>- getQueueTop changed logic for express boolean determination</li><li>- packed casted to uint128</li></ul></li><li>- <b>RocketMegapoolDelegate:</b> <a href="https://www.diffchecker.com/cWhwiBV0/">https://www.diffchecker.com/cWhwiBV0/</a><ul style="list-style-type: none"><li>- soonestDrawableEpoch removal</li><li>- implementation of nodeQueuedBond and userQueuedCapital</li><li>- implement edge-case of getNewValidatorBondRequirement</li></ul></li></ul>

- increase of nodeQueuedBond and userQueuedCapital during validator queue
  - dequeue logic changed
  - reduceBond logic changed
  - assignFunds logic changed
  - stake logic changed
  - validatorIndex fully removed
  - dissolveValidator logic changed
  - distributeAmount logic changed
  - calculateRewards logic changed
  - notifyExit logic changed
  - notifyFinalBalance logic **fully** changed
- 
- **RocketMegapoolManager:**  
<https://www.diffchecker.com/pLskohaO/>
    - implement SlotProof usage
    - implement Pubkey validation
    - recency implementation
    - pubkey uniqueness logic within addValidator
    - stake function hardening
      - \_slotTimestamp / \_slotProof
      - ensure validator state correctness
    - dissolve function hardening
      - \_slotTimestamp / \_slotProof
      - ensure validator state correctness
    - notifyExit hardening
      - \_slotTimestamp / \_slotProof
    - notifyNotExit hardening
      - \_slotTimestamp / \_slotProof
    - challengeExit hardening
      - Megapool existence check
    - notifyFinalBalance hardening
      - \_slotTimestamp / \_slotProof
      - verifyValidator
      - validatorIndex validation

- **BeaconStateVerifier:**

<https://www.diffchecker.com/b19AOstN/>

- implement genesisWitness
- verifyValidator/Withdrawal same as before
- verifySlot new
- BlockRoot directly fetched from slotTimestamp
  - before it was from slot -> slotTimestamp -> BlockRoot
- **RocketNodeDeposit:**  
<https://www.diffchecker.com/kJSDs2fB/>
  - hardened increaseDepositCreditBalance
  - \_deposit control-flow including pubkey to forward to MegapoolManager
- **RocketMegapoolPenalties:**  
<https://www.diffchecker.com/BdYxvZZi/>
  - switch from block to time
  - incorporate maxPenalty check
  - interact with RocketNetworkSnapshotsTime contract
  - externalize penaltyThreshold
- **RocketNetworkRevenues:**  
<https://www.diffchecker.com/aUYeP7Ni/>
  - onlyProtocolDAO modifier
  - initialize everything with timestamp instead of block.number
  - share return values based on timestamp instead of block.number
  - split calculation uses timestamp instead of block.number
  - \_calculateSplit enforces values to exist
  - implementation of averageCapitalRatio
- **RocketMegapoolStorageLayout:**  
<https://www.diffchecker.com/2PJukR6b/>
  - adjustment of storage variables

	<ul style="list-style-type: none"> <li>- <b>RocketNetworkPenalties:</b>  <a href="https://www.diffchecker.com/b3NH0bhy/">https://www.diffchecker.com/b3NH0bhy/</a> <ul style="list-style-type: none"> <li>- switch from block to days</li> <li>- allow for execution even if threshold is met exactly</li> </ul> </li>   <li>- <b>RocketMerkleDistributorMainnet:</b>  <a href="https://www.diffchecker.com/suB70GLb/">https://www.diffchecker.com/suB70GLb/</a> <ul style="list-style-type: none"> <li>- implement treeVersion</li> <li>- incorporate modular claiming [version0/version1]</li> </ul> </li>   <li>- <b>RocketRewardsPool:</b>  <a href="https://www.diffchecker.com/8tCtXOzw/">https://www.diffchecker.com/8tCtXOzw/</a> <ul style="list-style-type: none"> <li>- include treeVersion into MerkleDistributor.relayRewards</li> </ul> </li>   <li>- <b>RocketUpgradeOneDotFour:</b> Full audit</li> </ul> <p>The total amount 27 of issues has been newly identified within this review:</p> <p>4 High severity  6 Medium severity  8 Low severity  9 Informational severity</p> <p>All previously “Open” marked issues have been updated with their new status in the “Resolution 2 / Follow-Up” column.</p> <p>Please carefully read the “Security Advice - Follow-Up Audit”</p>
Resolution 3	This resolution round validates the fixes which stem from issues that have been identified during resolution 2. During the last resolution, the following contracts have been reviewed [partially or as diff]:

- RocketUpgradeOneDotFour
- RocketRewardsPool
- RocketMerkleDistributorMainnet
- RocketNetworkPenalties
- RocketMegapoolStorageLayout
- RocketNetworkRevenues
- RocketMegapoolPenalties
- RocketNodeDeposit
- BeaconStateVerifier
- RocketMegapoolManager
- RocketMegapoolDelegate
- DepositPool
- RocketDAOProtocolSettingsMegapool

The commit which includes all aggregated fixes is the following:

<https://github.com/rocket-pool/rocketpool/commit/59f1a2933ef26e6dc88bef21be0d41aac2e0221f>

The following files have changes **[inside the auditing scope]**:

- RocketDAOProtocolSettingsNode
- RocketDAOProtocolSettingsMegapool
- RocketDepositPool
- RocketNetworkRevenues
- RocketMegapoolDelegate
- RocketMegapoolManager

The following files have changes **[outside the auditing scope]**

- RocketDAOProtocolSettingsProposals [RPIP-64]
- RocketUpgradeOneDotFour [RPIP-64]

## 2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed resolution	Open
High	26	23	1	2		
Medium	30	12	3	15		
Low	32	6		24		2
Informational	56	19	1	36		
Governance						
<b>Total</b>	<b>144</b>	<b>60</b>	<b>5</b>	<b>77</b>		<b>2</b>

### 2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior

### 3. Detection

#### Security Advice - Initial Audit

##### a) Re-audit recommendation

Some modules will require extensive changes and potential refactoring. These components should undergo a *complete* re-audit, not only by BailSec but also by an independent second party.

A single resolution round would not provide the level of assurance needed to maintain high accuracy, as refactors typically introduce new and hard-to-detect vulnerabilities. There will be legitimate concerns of unidentified issues, including those that can result in loss of funds.

##### b) Upgrading live code during Saturn

Several issues have already been identified in live contracts and additional findings are detailed in the report. Addressing at least the high-severity issues will require upgrades to parts of the live codebase that was never meant to be upgraded during saturn.

This inclusion of previously immutable components in the Saturn upgrade introduces an additional layer of risk that must be acknowledged and mitigated through strict upgrade procedures and pre-deployment validation.

##### c) Confidence and residual risk

It cannot be stated with full confidence that all existing issues have been found. While BailSec likely uncovered substantially more vulnerabilities than all other audit firms combined [which is historically proven], there is still a probability that hidden flaws remain. This uncertainty must be transparently communicated and factored into the project's overall risk management strategy.

##### d) Operational monitoring and mitigation

It is highly recommended to implement real-time monitoring and alerting. Many flows depend on consensus-layer epochs, meaning that if a malicious action occurs, there is a detection window during which defensive measures can still be taken. Such monitoring is only effective if proxy delegations to outdated Megapool implementations are completely disallowed. Additionally, the protocol should consider introducing a DAO-controlled *pause mechanism* for Megapool-related interactions to allow rapid response in emergencies.

##### e) Non-upgradeable contract concerns

There are bugs present in contracts that were originally not intended to be updated. It needs

to be clearly determined whether these will be fixed and included in the upgrade cycle. Every deviation from the original assumptions introduces new safety concerns.

#### f] Megapool and network balance auditing

Finally, it is recommended to conduct a focused audit on the mechanism that determines the Megapool's impact on the overall ETH network balance. This verification should confirm that the accounting, timing, and data aggregation processes are robust, precise, and resistant to manipulation or measurement drift.

## Security Advice - Follow-Up Audit

Bailsec has conducted a follow-up audit based on the changes which are highlighted in the “Resolution 2/Follow-Up” column. The specified commit was considered as **finalized by other audit providers** and thus a **high confidence of deployment-correctness was expected**.

However, during our follow-up review we found multiple new issues which allowed for **stealing of funds and DoS of the protocol** which **decreases the confidence in a vulnerability-free code**, even after these have been fixed. This comes naturally due to code complexity and exponentiality of state transitions in the Megapool architecture.

While we have applied all our resources to this task, we can not say with high-confidence that there are no critical issues left. We recommend another full protocol audit by a third-party before deployment and a robust bug bounty program.

We highly recommend keeping the post-audit changes to a minimum and only fixing issues which are of a high severity. It must be highlighted that any potential change (even if minimal) can and will introduce new edge-cases.

## Global

Issue_01	Edge-case in voting mechanism in scenario of member removal
Severity	Low
Description	<p>Multiple voting mechanisms rely on a quorum which is based on the member count. If a member has voted, this vote will still count towards the direction - even if the member was removed.</p> <p>However, the removal of the member will inherently impact the new quorum as it is now less than expected.</p>
Recommendations	Consider keeping this in mind.
Comments / Resolution	Acknowledged.

## DAO/node/settings

### RocketDAONodeTrustedSettings

The `RocketDAONodeTrustedSettings` contract is the base contract which is inherited by:

- `TrustedSettingsMembers`
- `TrustedSettingsMinipools`
- `TrustedSettingsProposals`
- `TrustedSettingsRewards`

It exposes getter and setter functions for uint and bool.

**This contract has not been changed for the Saturn upgrade.**

Core Invariants:

INV 1: Only the `DAONodeTrustedProposals` contract is allowed to call `setSettingUint` and `setSettingBool`

Privileged Functions

- `setSettingUint`
- `setSettingBool`

No issues found.

## RocketDAO`NodeTrustedSettingsMembers`

The `RocketDAONodeTrustedSettingsMembers` contract inherits the `RocketDAONodeTrustedSettings` contract and exposes getter and setter functions for specific variables.

The `settingNameSpace` relates to `dao.trustednodes.setting.members` and the variables are related to the trusted members module.

**This contract has not been changed for the Saturn upgrade.**

**Core Invariants:**

INV 1: `members.quorum` must never be larger than 0.9 ETH and must be larger than 0 ETH

**Privileged Functions**

- `setSettingUint`

No issues found.



## RocketDAONodeTrustedSettingsMinipool

The `RocketDAONodeTrustedSettingsMinipool` contract inherits the `RocketDAONodeTrustedSettings` contract and exposes getter and setter functions for specific variables.

The `settingNameSpace` relates to `dao.trustednodes.setting.minipools` and the variables are related to minipools.

**This contract has not been changed for the Saturn upgrade.**

Core Invariants:

INV 1: The scrubPeriod must be less than the launch timeout - 1 hour

Privileged Functions

- `setSettingUint`

No issues found.

## RocketDAONodeTrustedSettingsProposals

The `RocketDAONodeTrustedSettingsProposal` contract inherits the `RocketDAONodeTrustedSettings` contract and exposes getter and setter functions for specific variables.

The `settingNameSpace` relates to `dao.trustednodes.setting.proposals` and the variables are related to trusted proposals

**This contract has not been changed for the Saturn upgrade.**

### Privileged Functions

- none

No issues found.



## RocketDAONodeTrustedSettingsRewards

The `RocketDAONodeTrustedSettingsRewards` contract inherits the `RocketDAONodeTrustedSettings` contract and exposes getter and setter functions for specific variables.

The `settingNameSpace` relates to `dao.trustednodes.setting.rewards` and the only used storage placement is the `rewards.network.enabled boolean` which is used by the `NodeManager's setRewardNetwork` function.

**This contract has not been changed for the Saturn upgrade.**

Core Invariants:

INV 1: reward network 0 can never be disabled

Privileged Functions

- `setSettingBool`

No issues found.

## DAO/node

### RocketDAONodeTrusted

The [RocketDAONodeTrusted](#) contract forms the base for the Trusted Nodes membership registry and implements various getter functions that reflect the current state of the registry, such as fetching whether an address is a Trusted Node or fetching the full member count. It implements a bootstrap mechanism which is already disabled in the current deployed version.

On top of that, an emergency [memberJoinRequired](#) function is called which allows for the permissionless addition of the own address as Trusted Node in the scenario when the minimum member count is undercut.

**This contract has not been changed for the Saturn upgrade.**

#### Appendix: Bootstrap Mechanism

During the bootstrap mechanism, the guardian address can do the following:

- Invite an address to join as a Trusted Node
- Setting uint within one of the four settings contracts
- Setting boolean within one of the four settings contracts
- Propose an upgrade
- Disable the bootstrap period

#### Core Invariants:

INV 1: Only the guardian can call all bootstrap functions

INV 2: [memberJoinRequired](#) is only callable if the minimum membercount is undercut

#### Privileged Functions

- [bootstrapMember](#)
- [bootstrapSettingUint](#)
- [bootstrapSettingBool](#)
- [bootstrapUpgrade](#)
- [bootstrapDisable](#)

No issues found.

## RocketDAONodeTrustedActions

The [RocketDAONodeTrustedActions](#) contract exposes the interface for users to join, leave and kick nodes as well as a challenge mechanism which allows for removing a Trusted Node.

Once users have been invited, they can become a Trusted Node via the [actionJoin](#) function, by providing 1750e18 RPL tokens as a bond. This bond will just be deposited into the vault and can then be re-claimed upon exiting. It is important to mention that an invited node must join within 4 weeks of the proposal invitation.

It is important to note that a node can only leave if a previous leave proposal has been passed and the leave request must not have expired.

**This contract has not been changed for the Saturn upgrade.**

### Appendix: Challenge Mechanism

The [actionChallengeMake](#) function allows any Trusted Node [even any node, for a fee] to challenge another Trusted Node. This can be useful in case a Trusted Node is inactive. If then within the [challengeWindow](#), a challenged node has not called the [actionChallengeDecide](#) function - to prove activity - anyone can call this function and kick the node.

### Core Invariants:

INV 1: Cannot challenge multiple nodes without waiting challenge cooldown time

INV 2: [actionChallengeDecide](#) can only be called with an actual challenged node

INV 3: [actionJoinRequired](#) is only callable through [memberJoinRequired](#) control-flow

INV 4: [actionLeave](#) can only be called if proposal leave request was successful

INV 5: [actionLeave](#) must refund RPL bond to user

INV 6: [actionKick](#) must burn the RPL fine and refund the RPL leftover bond

INV 7: Non-Trusted-Nodes must provide ETH to call [actionChallengeMake](#)



INV 8: A challenged but active node must call `actionChallengeDecide` to prevent self from being kicked

#### Privileged Functions

- `actionJoinRequired`
- `actionKick`

Issue_02	ETH is locked in case of actionChallengeMake call by non trusted node
Severity	Medium
Description	<p>The <code>actionChallengeMake</code> function allows non-trusted nodes to call it by providing the <code>challengeCost</code> in ETH. The ETH value is simply provided as <code>msg.value</code>.</p> <p>However, it will just remain sitting in the contract and there is currently no way to withdraw it.</p> <p>Also an upgrade will not help either because it is not deposited to the RocketVault.</p> <p><b>This issue is live in the current deployed version.</b></p>
Recommendations	<p>Consider elaborating what should happen with the provided ETH and either implement a solution or simply not allow non trusted nodes to call this function.</p> <p>Depending on the solution, an audit of the full control-flow may be required.</p>
Comments / Resolution	<p>Acknowledged. The team mentioned that a fee is required in the scenario where a non-trusted node invokes that function and this is not considered as a vulnerability</p> <p>We fully agree with this mechanism as it is used to prevent abusive calls. However, we do not agree with the fact that it is acceptable that this fee just remains permanently locked in the contract. Indeed, one could argue that such a design decision would make sense for the native token [RPL] to make it deflationary.</p> <p>However, this is not true for ETH and we are of the opinion that it is a serious vulnerability if ETH remains permanently locked without any possibility for governance to claim it. We have downgraded the severity from high to medium.</p>

Issue_03	Node with outstanding invitation cannot be kicked
Severity	Medium
Description	<p>Rocket Pool's Trusted Node DAO ("Trusted Nodes") onboards members via an invite → join flow.</p> <p>A candidate is first invited by a successful proposal (recorded as an executed "invited" action time).</p> <p>The invitee must then accept by calling <code>actionJoin</code> within the DAO's configured action time window, at which point the contract: [1] pulls the required RPL bond, [2] records membership, and [3] adds the address to the member index.</p> <p>Members may later be removed by <code>actionKick</code> (proposal-driven) or via the challenge flow (<code>actionChallengeDecide</code>).</p> <p>Once an invite proposal has been executed, there is no mechanism to revoke that invitation before the invitee joins.</p> <p>An invitee who has not called <code>actionJoin</code> does not satisfy <code>onlyTrustedNode</code>, so attempts to kick/remove them revert at the modifier.</p> <p>Conversely, <code>_memberJoin</code> only checks that the "invited" timestamp exists and has not expired (action time), plus allowance for the bond; there is no "revoked" flag. Thus, after an invite has executed, the invitee has an irrevocable right to join until the action window closes.</p> <p>Because <code>actionJoin</code> immediately calls <code>_memberAdd</code> (which adds the member to the index and sets <code>member=true</code>) and there is no join-to-vote delay in this contract, the invitee can vote immediately after joining. Operationally, they can also time their join at the last moment of the window (after sentiment has shifted) and cast votes straight away.</p> <p>Now one can argue that another <code>proposalKick</code> proposal can be triggered, however, that would require going through the voting</p>

	<p>delay which means the proposal we intended to vote for, will be quicker to reach its votable state than the kick proposal.</p> <p><b>This issue is live in the current deployed version.</b></p>
<b>Recommendations</b>	Consider implementing logic that allows for removing an invitation. This logic needs to be fully audited.
<b>Comments / Resolution</b>	Acknowledged.

<b>Issue_04</b>	Lack of member.challenged.by reset during <code>actionChallengeDecide</code>
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>The <code>actionChallengeMake</code> function sets:</p> <pre>setAddress[keccak256[abi.encodePacked(daoNameSpace, "member.challenged.by", _nodeAddress)], msg.sender];</pre> <p>This state is not reset during <code>actionChallengeDecide</code>.</p>
<b>Recommendations</b>	Consider if this is intentional, if not, consider resetting this value.
<b>Comments / Resolution</b>	Acknowledged, the Rocketpool team mentions because this contract is not being targeted for upgrade in Saturn, they will consider it in a future upgrade.

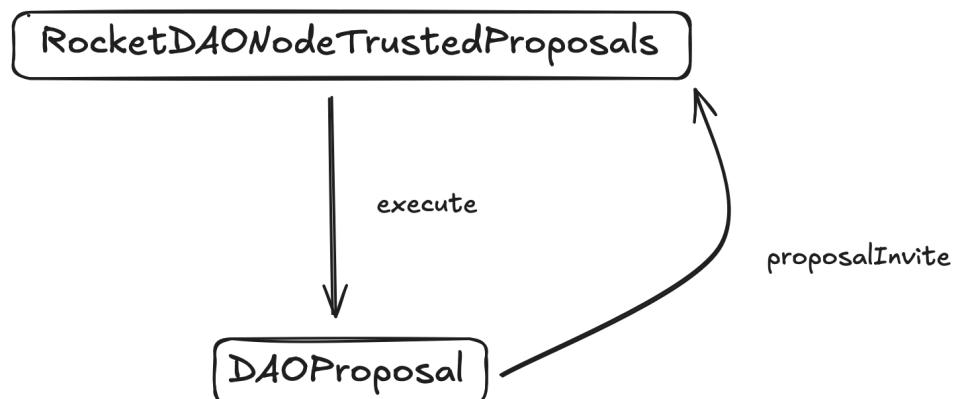
## RocketDAONodeTrustedProposals

The `RocketDAONodeTrustedProposals` contract implements the proposal and voting logic for Trusted Nodes as well as all important proposal interactions which are called by the `DAOProposal` contract upon successful execution of proposals.

This contract has not been changed for the Saturn upgrade.

### Appendix: Interaction with RocketDAOProposal

Within the whole DAO module, proposal executions are circular which means they are triggered outgoing from the specific proposal contract into the `DAOProposal` contract and then a callback towards the specific proposal contract is executed to trigger one of the proposal interaction functions:



### Core Invariants:

INV 1: Only Trusted Nodes can propose

INV 2: Only Trusted Nodes can vote on proposals

INV 3: Propose is only callable after cooldown time has passed

INV 4: Propose is only callable if minimum member count is satisfied

INV 5: A member can only vote for a proposal if he has joined before the proposal has been created

INV 6: All invitations must run through proposallInvite

#### Privileged Functions

- proposallInvite
- proposalLeave
- proposalKick
- proposalSettingUint
- proposalSettingBool
- proposalUpgrade

No issues found

## RocketDAONodeTrustedUpgrade

The [RocketDAONodeTrustedUpgrade](#) contract is responsible for the upgrade execution whenever an upgrade proposal has:

- a] passed and is executed
- b] upgradeDelay has passed
- c] upgrade is not veto'd

It is not allowed to upgrade the following contracts:

- RocketVault
- RocketTokenRETH
- RocketTokenRPL
- RocketTokenRPLFixedSupply
- CasperDeposit
- RocketMinipoolPenalty

This contract has been changed for the Saturn upgrade.

### Appendix: Veto Mechanism

Even when an upgrade proposal has passed and was executed, it can only be upgraded after the [upgradeDelay](#) has passed. Within this period, a proposal can be veto'd, via a proposal within the Security module. In case a veto proposal has passed, the upgrade cannot be executed.

### Appendix: Upgrade Proposal States

An upgrade proposal can have the following states:

- Veto'd: Whenever a veto proposal has been executed. The upgrade can not be executed anymore
- Executed: The upgrade has been fully executed
- Pending: The upgrade is still in the delay period
- Succeeded: The delay period has successfully passed without any veto and the execution has not happened yet.

## Core Invariants:

INV 1: Upgrade can only be executed after upgradeDelay has passed

INV 2: Veto is only possible while the upgrade is within the upgradeDelay period

INV 3: Multiple contracts cannot be upgraded

INV 4: contract.address is the source of truth for determining which contract is the current one

## Privileged Functions

- upgrade
- veto
- bootstrapUpgrade

<b>Issue_05</b>	Lack of expiration state for upgrade allows for executing stale upgrades
<b>Severity</b>	Medium
<b>Description</b>	<p>In the current mechanism, it is allowed to execute successful upgrade proposals anytime, without any restriction.</p> <p>This allows for executing upgrade proposals far later after they succeeded.</p> <p>This is especially notable because there may be newer upgrades that have already been executed and now the execution of a stale upgrade will break the setup.</p> <p><b>This issue is live in the current deployed version.</b></p>
<b>Recommendations</b>	Consider implementing an expiry mechanism.
<b>Comments / Resolution</b>	Acknowledged.

Issue_06	Non-callable <code>_addContract</code> after <code>_addABI</code> will permanently burn <code>_name</code>
Severity	Low
Description	<p>The <code>_addABI</code> function requires that the corresponding contract is non-existing and at the same time the <code>_addContract</code> function requires the ABI to be non-existent.</p> <p>However, execution requires for both functions to be called for a contract to be added, but this logic is contradictory.</p> <p><b>This issue is live in the current deployed version.</b></p>
Recommendations	Consider removing these restrictions.
Comments / Resolution	<p>Acknowledged. This is considered as a non-issue as per Rocketpool because the main way to add new contracts is via the <code>addContract</code> control-flow which correctly adds an address and ABI at the same time.</p> <p>While this control-flow execution will indeed work without any side-effects, the above mentioned issue is still technically valid. The severity has been downgraded from medium to low with the provided context.</p>

## DAO/protocol/settings

### RocketDAOProtocolSettings

The [RocketDAOProtocolSettings](#) contract is the base contract which is inherited by:

- ProtocolSettingsAuction **[out-of-scope]**
- ProtocolSettingsDeposit
- ProtocolSettingsInflation
- ProtocolSettingsMegapool
- ProtocolSettingsMinipool
- ProtocolSettingsNetwork
- ProtocolSettingsNode
- ProtocolSettingsProposals
- ProtocolSettingsRewards
- ProtocolSettingsSecurity

It exposes getter and setter functions for uint, bool, address and batch addresses

**This contract has not been changed for the Saturn upgrade.**

Core Invariants:

INV 1: Only the DAOProtocolProposals contract can call all the privileged functions

#### Privileged Functions

- setSettingUint
- setSettingBool
- setSettingAddress
- setSettingAddressList

No issues found.



## RocketDAOProtocolSettingsDeposit

The [RocketDAOProtocolSettingsDeposit](#) contract inherits the RocketDAOProtocolSettings contract and exposes getter and setter functions for specific variables.

The `settingNameSpace` relates to `dao.protocol.setting.deposit` and the variables are related to the `DepositPool` and `NodeManager`.

**This contract has been changed for the Saturn upgrade.**

Core Invariants:

INV 1: `deposit.fee` cannot be  $\geq 0.01$  ETH

INV 2: `express.queue.rate` cannot be zero

Privileged Functions

- `setSettingUint`

No issues found.



## RocketDAOProtocolSettingsInflation

The [RocketDAOProtocolSettingsInflation](#) contract inherits the [RocketDAOProtocolSettings](#) contract and exposes getter and setter functions for specific variables.

The `settingNameSpace` relates to `dao.protocol.setting.inflation` and the variables are related to the [TokenRPL](#) contract and the inflation mechanics.

**This contract has not been changed for the Saturn upgrade.**

Core Invariants:

INV 1: `rpl.inflation.interval.start` can only be changed if inflation has not yet started

INV 2: `rpl.inflation.interval.rate` cannot be  $> 0.01$  ETH larger than the previous rate and cannot be  $< 1$

Privileged Functions

- `setSettingUint`

No issues found.



## RocketDAOProtocolSettingsMegapool

The [RocketDAOProtocolSettingsMegapool](#) contract inherits the [RocketDAOProtocolSettings](#) contract and exposes getter and setter functions for specific variables.

The settingNameSpace relates to dao.protocol.setting.megapool and the variables are related to the Megapool module.

**This contract is new for the Saturn upgrade.**

**Core Invariants:**

INV 1: megapool.time.before.dissolve must be  $\geq$  2 days

INV 2: maximum.megapool.eth.penalty must be  $\geq$  300 ETH

INV 3: notify.threshold must be  $\geq$  2 hours

INV 4: late.notify.fine must be  $<$  0.5 ETH

INV 5: user.distribute.window.length must be between 1 and 30 days

**Privileged Functions**

- setSettingUint

<b>Issue_07</b>	Off-by-one error within late.notify.fine
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>The following comment is next to the setting for late.notify.fine:</p> <pre><i>require_value &lt; 0.5 ether, "Fine must be less than or equal to 0.5 ETH"; // Per RPIP-72</i></pre> <p>This comment is wrong, as it is not allowed to be equal 0.5 ETH.</p>
<b>Recommendations</b>	Consider either fixing the comment or the off-by-one error.
<b>Comments / Resolution</b>	<p>Resolved, the requirement has been changed as follows:</p> <pre><i>} else if [settingKey == keccak256[bytes("late.notify.fine")]] {     require_value &gt;= 0.01 ether &amp;&amp; _value &lt;= 0.5 ether, "Value must be &gt;= 0.01 ETH &amp; &lt;= 0.5 ETH";</i></pre> <p>It is now allowed to implement a late.notify.fine up to 0.5 ETH</p>



## RocketDAOProtocolSettingsMinipool

The `RocketDAOProtocolSettingsMinipool` contract inherits the `RocketDAOProtocolSettings` contract and exposes getter and setter functions for specific variables.

The `settingNameSpace` relates to `dao.protocol.setting.minipool` and the variables are related to the Minipool module.

**This contract has been changed for the Saturn upgrade.**

Core Invariants:

INV 1: `minipool.launch.timeout` must be larger/equal 12 hours and larger/equal `scrubPeriod + 1 hour`

INV 2: `minipool.maximum.penalty.count` must be larger/equal 2500

Privileged Functions

- `setSettingUint`

No issues found.

## RocketDAOProtocolSettingsNetwork

The [RocketDAOProtocolSettingsNetwork](#) contract inherits the [RocketDAOProtocolSettings](#) contract and exposes getter and setter functions for specific variables.

The `settingNameSpace` relates to `dao.protocol.setting.network` and the variables are used throughout the whole [RocketPool](#) architecture.

**This contract has been changed for the Saturn upgrade.**

### Appendix: Shares

- a) `network.node.commission.share.security.council.adder`: Deducted from `voterShare`, added on top of `nodeShare`. Serves as rebalancing mechanism
- b) `network.node.commission.share`: Determines `nodeShare` amount
- c) `network.voter.share`: Determines `voterShare` amount
- d) `network.pdao.share`: Determines `protocolDAOShare` amount

### Appendix: AllowListedController

The [AllowListedController](#) set is related to a specific setup which considers the `ProtocolSettings.getSettingAddressList` function with the key: `network.allow.listed.controllers`. Addresses within this list are allowed to call

- `setNodeShareSecurityCouncilAdder`
- `setNodeCommissionShare`
- `setVoterShare`
- `setProtocolDAOShare`

### Core Invariants:

INV 1: Only addresses within the `network.allow.listed.controllers` address list are allowed to change share settings

INV 2: `getEffectiveNodeShare` must combine `network.node.commission.share` and `network.node.commission.share.security.council.adder`

INV 3: `getEffectiveVoterShare` must deduct `network.node.commission.share.security.council.adder` from `network.voter.share`

INV 4: setSettingUint must have multiple constraints with regards to variable settings

INV 5: network.node.commission.share.security.council.adder must not be larger than network.voter.share

#### Privileged Functions

- setSettingUint
- setNodeShareSecurityCouncilAdder
- setNodeCommissionShare
- setVoterShare
- setProtocolDAOShare

No issues found.

## RocketDAOProtocolSettingsNode

The RocketDAOProtocolSettingsNode contract inherits the RocketDAOProtocolSettings contract and exposes getter and setter functions for specific variables.

The settingNameSpace relates to dao.protocol.setting.node and the variables are used throughout the whole RocketPool architecture.

**This contract has been changed for the Saturn upgrade.**

### Privileged Functions

- setSettingUint

No issues found.



## RocketDAOProtocolSettingsProposals

The [RocketDAOProtocolSettingsProposals](#) contract inherits the [RocketDAOProtocolSettings](#) contract and exposes getter and setter functions for specific variables.

The `settingNameSpace` relates to `dao.protocol.setting.proposals` and the variables are used within the DAO Protocol Module.

**This contract has not been changed for the Saturn upgrade.**

Core Invariants:

INV 1: `setSettingUint` must incorporate various constraints

Privileged Functions

- `setSettingUint`

No issues found

## RocketDAOProtocolSettingsRewards

The [RocketDAOProtocolSettingsRewards](#) contract inherits the [RocketDAOProtocolSettings](#) contract and exposes getter and setter functions for specific variables.

The `settingNameSpace` relates to `dao.protocol.setting.rewards` and the variables are used within the Rewards module.

This contract has not been changed for the Saturn upgrade.

Core Invariants:

INV 1: `_trustedNodePercent + _protocolPercent + _nodePercent` must be equal to `1e18`

Privileged Functions

- `setSettingRewardsClaimers`

<b>Issue_08</b>	<code>reduced.bond</code> can be set with extended digits [FOLLOWUP]
<b>Severity</b>	Medium
<b>Description</b>	<p>The <code>reduced.bond</code> value can be set for example to <code>3.00000000000005e18</code>. This would break any further validator addition due to the following check within the DepositPool:</p> <pre>require[_bondAmount % milliToWei == 0, "Invalid supplied amount"];</pre>
<b>Recommendations</b>	Consider not allowing such a setting.
<b>Comments / Resolution</b>	Resolved.



## RocketDAOProtocolSettingsSecurity

The [RocketDAOProtocolSettingsSecurity](#) contract inherits the [RocketDAOProtocolSettings](#) contract and exposes getter and setter functions for specific variables.

The `settingNameSpace` relates to `dao.protocol.setting.security` and the variables are used within the Security module.

**This contract has been changed for the Saturn upgrade.**

Core Invariants:

INV 1: `setSettingUint` must incorporate various constraints

Privileged Functions

- `setSettingUint`

No issues found.

## DAO/protocol

### RocketDAOProtocol

The [RocketDAOProtocol](#) contract handles the bootstrap mechanism for the Protocol module, allowing the guardian to set various variables. At the current state of the protocol, the bootstrap mechanism is disabled and will not be used anymore.

This contract has been changed for the Saturn upgrade.

#### Privileged Functions

- bootstrapSettingMulti
- bootstrapSettingUint
- bootstrapSettingBool
- bootstrapSettingAddress
- bootstrapSettingAddressList
- bootstrapSettingClaimers
- bootstrapSpendTreasury
- bootstrapSpendTreasuryNewContract
- bootstrapSpendTreasuryUpdateContract
- bootstrapSecurityInvite
- bootstrapSecurityKick
- bootstrapDisable
- bootstrapEnableGovernance

No issues found.



## RocketDAOProtocolActions

The [RocketDAOProtocolActions](#) contract is an empty contract. It is currently not used.

This contract has not been changed for the Saturn upgrade.

### Privileged Functions

- none

## RocketDAOProtocolProposal

The [RocketDAOProtocolProposal](#) contract allows any registered node to submit a proposal, including a root which is meant to reflect the current state of the protocol with the correct voting power. It is directly interacting with the [DAOProtocolVerifier](#) which implements an optimistic fraud-proof system that allows challenging invalid proposal states and verifies votes.

This contract has not been changed for the Saturn upgrade.

### Appendix: Proposal Lifecycle

A Protocol DAO proposal begins when a registered node invokes the [propose](#) function.

The contract first ensures on-chain governance is enabled and that `blockNumber` is recent enough and  $\geq$  the DAO enable block. It then sums the pollard [treeNodes] to obtain the total voting power at that snapshot, derives the phase timings and quorums from [RocketDAOProtocolSettingsProposals](#) [vote delay, phase 1 duration, phase 2 duration, execution window, proposal quorum, veto quorum], and stores a new proposal entry under the `dao.protocol.proposal.*` namespace.

Immediately after, it calls the Verifier's [submitProposalRoot](#) to anchor the root: the Verifier checks pollard sizing against the network's node count at `blockNumber`, locks the proposer's RPL bond [via [RocketNodeStaking.lockRPL](#)], stores the proposal's root and parameters [node count, bonds, challenge period], and marks the root index as "responded".

From here the proposal is Pending until its start time elapses. The lifecycle then splits into two voting phases:

a) Phase 1 [ActivePhase1]: Any registered node may cast a vote with a proof: `vote[proposalId, direction, votingPower, nodeIndex, witness]`.

The Verifier validates the proof of voting power at the proposal's snapshot using the stored root; if valid, the proposal contract records the vote in the appropriate bucket [for, against, against with veto, abstain] and remembers that this voter participated in phase 1.

b) Phase 2 [ActivePhase2]: Any registered node may *override* their own vote relative to their delegate's phase-1 direction: `overrideVote[proposalId, direction]`. The proposal contract looks up the caller's delegate and voting power [from [RocketNetworkVoting](#) at the snapshot block], optionally reverses the delegate's contribution [if different], and then applies the caller's own vote. This gives individual operators a second-chance control over their representation.

After phase 2 has elapsed, there are multiple distinct outcomes:

- a) If veto votes reach the veto quorum, the proposal is Vetoed. The proposal can then be finalised: `finalise[proposalId]` [once], which instructs the Verifier to burn the proposer's bond [`unlock + burn` via the staking contract]
- b) If not veto'd, the contract computes `totalVotes = for + against + abstained`.  
If `totalVotes` is below quorum, the proposal is `QuorumNotMet`.  
If quorum is met and `against ≥ for`, it is `Defeated`.  
If quorum is met and `for > against`, it `Succeeds` provided the current time is before `expires`.

After expiration, a successful but unexecuted proposal becomes `Expired`.

A *succeeded* proposal may be executed at any time before *expires* by calling `execute[proposalId]`: the contract marks it `executed = true` and then performs the payload on `rocketDAOProtocolProposals`. An executed proposal is executed thereafter.

## Appendix: Proposal States

**Pending:** The proposal has been created but voting has not started yet

**ActivePhase1:** The proposal is within the first voting period, where votes must be submitted with Merkle proofs of voting power.

**ActivePhase2:** The proposal is within the second voting period, where node operators can override their delegate's Phase-1 votes directly.

**Destroyed:** The proposal was invalidated by the fraud-proof mechanism and is permanently removed from the lifecycle.

**Vetoed:** Enough veto votes were cast, so the proposal is stopped and the proposer's bond can be burned.

**QuorumNotMet:** The voting period finished, but the total votes did not meet the minimum quorum requirement.

**Defeated:** The voting reached quorum, but the votes against were greater than or equal to the votes for.

**Succeeded:** The proposal passed quorum, had more votes for than against, and is within its execution window, but has not yet been executed.

**Expired:** The proposal succeeded but was not executed before the expires timestamp, and thus can no longer be enacted.

**Executed:** The proposal succeeded and its payload has been executed on the target DAO contract.

#### Core Invariants:

INV 1: A proposal's \_blockNumber must always be < block.number [cannot use current or future block].

INV 2: A proposal's \_blockNumber + proposalMaxBlockAge must always be > block.number [snapshot cannot be too old].

INV 3: A proposal's start timestamp must be strictly > block.timestamp at creation.

INV 4: Each proposal must have exactly one proposer address stored, which cannot be changed after creation.

INV 5: A node can only vote once per proposal

INV 6: Phase 1 votes can only be cast when getState[proposalID] == ActivePhase1.

INV 7: Phase 2 overrides can only be cast when getState[proposalID] == ActivePhase2.

INV 8: If overriding a delegate's Phase 1 vote, the delegate's recorded votes must be reduced exactly by the voter's voting power.

INV 9: votes.for + votes.against + votes.abstained must equal the sum of all receipts for a given proposal.

INV 10: Veto status [getVetoed] must equal votes.veto  $\geq$  vetoQuorum.

INV 11: Total proposals [getTotal] must always increase by exactly one per new proposal.

#### Privileged Functions

- destroy

<b>Issue_09</b>	<b>overrideVote</b> can be used to change voting direction
<b>Severity</b>	Medium
<b>Description</b>	<p>Within the first period, it is not allowed to change the voting direction. This has different reasons but it is generally considered to only allow for voting once. This can be bypassed by a user simply via voting during period 1 with address A that is the delegatee from address B. During period 2, address B can then vote and thus the user was able to switch voting direction while that should not be allowed.</p> <p><b>This issue is live in the current deployed version.</b></p>
<b>Recommendations</b>	A fix for this issue is non-trivial as it would require refactoring the whole process.
<b>Comments / Resolution</b>	<p>Acknowledged. Rocketpool added the following comment:</p> <p>"This is acceptable. A delegate should not be able to switch their vote during phase 1. The reason for this is we don't want a delegatee to vote one direction at the start of phase 1 indicating they are voting in a certain direction, then changing it at the last minute. This may trick delegators into thinking their delegatee is voting in the direction they desire and they do not have to override the vote when the time comes. We don't have any concerns with a voter switching their own voting power in phase 2."</p>

<b>Issue_10</b>	Lack of cancellation possibility for proposals
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>Users can create new proposals via the <code>propose</code> function while locking the corresponding RPL amount.</p> <p>There is currently no such functionality to cancel a proposal and get back the bond, which can inevitably result in a user losing his bond if the provided state is wrong.</p>
<b>Recommendations</b>	Consider acknowledging this issue. Introducing such a change will introduce new state transitions and potentially new bugs.
<b>Comments / Resolution</b>	<p>Acknowledged, Rocketpool added the following comment:</p> <p>“This is intentional. Making a proposal with an invalid state is a punishable offence”</p> <p>It is important to note that this issue was not in the context of a proposal with an invalid state but rather in context of a regular proposal where the proposer switched his decision.</p>



## RocketDAOProtocolProposals

The [RocketDAOProtocolProposals](#) contract handles the execution of successful proposals. It essentially exposes all functions that can be triggered by the [RocketDAOProtocolProposal](#) contract upon the execute method. It is not used for any other purpose.

**This contract has been changed for the Saturn upgrade.**

Core Invariants:

INV 1: Only the RocketDAOProtocolProposal contract must be allowed to call privileged functions.

### Privileged Functions

- proposalSettingMulti
- proposalSettingUint
- proposalSettingBool
- proposalSettingAddress
- proposalSettingAddressList
- proposalSettingRewardsClaimers
- proposalTreasuryOneTimeSpend
- proposalTreasuryNewContract
- proposalTreasuryUpdateContract
- proposalSecurityInvite
- proposalSecurityKickMulti
- proposalSecurityReplace

No issues found.

## RocketDAOProtocolVerifier

The [RocketDAOProtocolVerifier](#) contract implements a complex optimistic fraud-proof system which serves as a safety mechanism against invalid proposal creations and votes with the incorrect voting power.

**This contract has not been changed for the Saturn upgrade.**

### Appendix: Tree Structure

During the [submitProposalRoot](#) function, an array of Nodes is provided which is then used to compute the root via the [computeRootFromNodes](#) function. These Nodes reflect the VP for each node (based on all mapped delegations).

The root is simply built via hashing each node upwards, until the final root is created by hashing its lower two nodes. These nodes are built via the node x delegate matrix at the base layer. The full tree can be illustrated as follows:



And below the row layer, are simply the hashes which are built upwards from the base layer, while the base layer is a [delegateIndex, nodeIndex] matrix.

The row layer is of size  $N = \text{nodes}$  and the base layer is of size  $N \times N$ . Notably, in the current state of the RocketPool protocol, where the network has a large number of nodes, any proposer must always submit 32 nodes at depth 5 which are the pollard slice above the row layer, each being a hash+sum aggregate of multiple row nodes.

### Appendix: Challenge State

The challenge state reflects the current state of a challenge, in the following format:

- 255 - 248: Challenge state
- 247 - 184: timestamp for [createChallenge](#)
- 183 - 24: Address of challenger
- 24 - 0: zero

These are the potential states:

- unchallenged: index has not been challenged
- challenged: index has been challenged but not responded
- responded: proposer has responded to challenge
- paid: index has been settled

#### Appendix: Challenge / Submit race

The challenge lifecycle can contain multiple rounds of `createChallenge` → `submitRoot` → `createChallenge` → ....

This follows from the Merkle tree's depth and the fact that each `submitRoot` only resolves the currently challenged subtree by submitting a next-depth pollard and updating the commitment at that index; thereafter, challengers may open a new challenge against any eligible next-depth index within that subtree (as constrained by `getPollardRootIndex` and the proof-length check).

The process proceeds in `depthPerRound` increments and therefore requires at most  $\lceil \text{totalDepth}/\text{depthPerRound} \rceil$  rounds to descend to the base-layer leaves (node × delegate), where correctness is checked directly against on-chain values via `verifyLeaves`.

It is important to highlight that the `depthPerRound` value [5] determines how far down the tree a challenge must descend in each round. Whenever a challenger targets a given index, the proposer is required to provide a pollard slice that extends exactly `depthPerRound` levels deeper than that index. This process repeats round by round: each new `submitRoot` resolves only the currently challenged subtree and replaces it with a new commitment one step deeper. In this way, `depthPerRound` fixes the maximum size of each proof (32 nodes) while bounding the number of rounds required to eventually reach the root layer and, if necessary, the base layer leaf nodes will be subject to direct on-chain verification (`verifyLeaves`). This is clamped by the final depth of the tree.

After challenge, the proposer must submit indexes at depth D below the challenged index. This is depending on the overall depth based on `nodeCount` but usually its 5 depths below the challenged, with each depth below increasing the required node reveals exponentially

- depth 6: reveal 2
- depth 7: reveal 4
- depth 8: reveal 8
- depth 9: reveal 16
- depth 10: reveal 32

## Appendix: Vote Verification

The `verifyVote` function checks that the voting power a voter claims is valid by reconstructing the Merkle path: it takes the voter's self-supplied leaf (`sum = votingPower, hash = keccak256(votingPower)`) and combines it with the provided witness nodes up to the root, then verifies that this reconstructed root matches the stored proposal root.

### Core Invariants:

INV 1: The row layer is always at depth `ceil_log2[nodeCount]`

INV 2: The row layer must be appended with zero paddings for the rounded up to  $2^D$  entries

INV 3: If `nodeCount > 32`, proposer still submits 32 entries at `depth = 5`, which is not row layer, this can be challenged down

INV 4: Anything above the row layer is an aggregate of the layer below, starting at the row layer

INV 5: Any challenging mechanism requires to reveal the layer below the proposer layer, starting with next depth [5] or clamps to the final depth

INV 6: The proposer will either submit the row layer [ $\leq 5$  depth] or the layer at `depth = 5` [ $\geq 5$  depth].

INV 7: The row layer always represents all nodes and their corresponding real (delegated) VP

INV 8: In the current contract state, there are always  $> 32$  registered nodes. It will thus always require to provide data at `depth = 5` which does not reflect the direct row layer

INV 9: When challenging, `_node` is one Node from the provided `_treeNode` (the corresponding index)

INV 10: Defining the index to challenge, it will be from 0 to 31 from the proposal layer, which is  $32 - 63$  (index based tree number)

INV 11: During `createChallenge`, the first index must always be between 32-63, as it is using the global index. The premise is that we have  $> 32$  registered nodes.

INV 12: The base layer contains row layer entries \* row layer entries (if we have 64 registered nodes, it contains 4096 entries)

INV 13: Within verifyVote, [nodeIndex](#) must represent the index of the node in the row layer.  
Distinct difference between tree index and node index

INV 14: It is only allowed to challenge a pending proposal

INV 15: The same index of a proposal [can only be challenged once](#)

INV 16: A defeated proposal [cannot be challenged](#)

INV 17: A proposal can be [defeated](#) if the proposer did not react to the challenge within [challengePeriod](#)

INV 18: [claimBondProposer](#) must only be allowed if proposer has [responded to index](#)

INV 19: claimBondProposer [burns a part](#) of the challengeBond

INV 20: claimBondChallenger is only allowed for the original challenger of an index

INV 21: Voting period must have passed during [claimBondProposer](#)

INV 22: It is [not possible to challenge an index at base layer](#)

INV 23: When challenging, previousIndex [getPollardRootIndex] must always be the one [which was previously responded](#)

INV 24: claimBondProposer can be called for any challenge [which has been responded](#)

INV 25: After a challenge is marked as PAID [challenger / proposer claimer], it can never be changed in state

#### Privileged Functions

- submitProposalRoot
- burnProposalBond

<b>Issue_11</b>	Race condition for increase of proposalBond/challengeBond can result in unexpected lock/loss
<b>Severity</b>	Medium
<b>Description</b>	<p>Whenever users submit a proposal or challenge a proposal, they need to lock a bond which can later be withdrawn by the owner or by the opponent party - depending on the outcome.</p> <p>Due to the fact that <code>proposalBond/challengeBond</code> are dynamic values, it can happen that these values are changed right before a transaction is executed, resulting in the wrong RPL amount being locked.</p> <p><b>This issue is live in the current deployed version.</b></p>
<b>Recommendations</b>	<p>Consider implementing slippage parameters or warning users if such a governance proposal is running.</p> <p>It has to be noted that if bugs are fixed in contracts that did not mean to be upgraded, the upgrade mechanism must be re-adjusted which further introduces safety concerns.</p>
<b>Comments / Resolution</b>	Acknowledged. Rocketpool considered the implication of this and deemed it acceptable.



## DAO/security

### RocketDAOSecurity

The [RocketDAOSecurity](#) contract forms the base for the Security membership registry and implements various getter functions that reflect the current state of the registry, such as fetching whether an address is a security member or fetching the full member count.

**This contract has not been changed for the Saturn upgrade.**

#### Privileged Functions

- none

No issues found.

## RocketDAOSecurityActions

The [RocketDAOSecurityActions](#) contract exposes the interface for users to join, leave and kick nodes from the security module. Users can only join once they have been invited beforehand and as long as the invitation has not expired.

Any invitation must go through the Protocol Proposal module. Likewise, kicking users must also have a successful proposal as a precondition.

The leave process requires first a call to [actionRequestLeave](#) which then must be followed by an [actionLeave](#) call after the leave time has been surpassed.

**This contract has not been changed for the Saturn upgrade.**

Core Invariants:

INV 1: Kicking user is only possible via a successful protocol proposal

INV 2: A user can only join if he has been invited before

INV 3: `_memberJoin` reverts if an invitation is expired

INV 4: Leaving from the module requires a two-step process

Privileged Functions

- `actionKick`
- `actionKickMulti`

No issues found.



## RocketDAO Security Proposals

The [RocketDAO Security Proposals](#) contract implements the propose and voting interface for members of the security modules as well as all important proposal interactions which are called by the [DAOProposal](#) contract upon successful execution of proposals, such as:

- proposalSettingUint
- proposalSettingBool
- proposalSettingAddress
- proposalInvite
- proposalKick
- proposalKickMulti
- proposalReplace

It is important to highlight that all setting functions can directly write to the `dao.protocol.setting` namespace and all its corresponding submodules:

- Deposit
- Inflation
- Megapool
- Minipool
- Network
- Node
- Proposals
- Rewards
- Security

**This contract has been changed for the Saturn upgrade.**

Core Invariants:

INV 1: Only security members can propose or vote

INV 2: `proposalSettingUint`/`proposalSettingBool`/`proposalSettingAddress` can only be called with valid settings

## Privileged Functions

- proposalSettingUint
- proposalSettingBool
- proposalSettingAddress
- proposalInvite
- proposalKick
- proposalKickMulti
- proposalReplace

<b>Issue_12</b>	Node with outstanding invitation cannot be kicked
<b>Severity</b>	Medium
<b>Description</b>	<p>This issue is severe because the <code>getMemberJoinedTime</code> is compared with <code>DAOProposal.getCreated</code>, hence a user can join exactly before a proposal has been created. Now one can argue that another <code>proposalKick</code> proposal can be triggered, however, that would require to go through the voting delay which means the proposal we intended to vote for, will be quicker to reach its votable state than the kick proposal.</p> <p><b>This issue is live in the current deployed version.</b></p>
<b>Recommendations</b>	Consider implementing logic which allows for a DAO proposal to revoke the invitation state of an address.
<b>Comments / Resolution</b>	Acknowledged.

Issue_13	Lack of RPLToken.inflationMintTokens call in case of rpl.inflation.interval.rate update
Severity	Informational
Description	<p>Currently, there are some insufficient settings/external calls in case of specific variable changes.</p> <p>such as the issue described in the title and:</p> <ul style="list-style-type: none"> <li>- pDAOShare, voterShare, nodeShare does not execute external call to NetworkRevenues</li> <li>- node.voting.power.stake.maximum - not pushed into NetworkSnapshots</li> </ul> <p>Please note that this issue is only informational due to the <code>onlyValidSetting</code> modifier and the way how the settings were determined within the constructor of the <code>ProtocolSettingsSecurity</code> contract. If there will be more valid settings introduced in a future upgrade, it is advised to cross-check sanity checks and external calls.</p>
Recommendations	Consider keeping this in mind for future upgrades.
Comments / Resolution	Acknowledged.

<b>Issue_14</b>	Lack of sanity checks in settings
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>Various functions write to <code>dao.protocol.setting.xxx</code> which is similar to the functionality of contracts within <code>/protocol/settings</code>.</p> <p>However, the same sanity checks which are exposed within these contracts are missing here. Just one instance for example:</p> <pre>if [settingKey == keccak256(abi.encodePacked("deposit.fee"))] {     require[_value &lt; 0.01 ether, "Fee must be less than 1%"]; }</pre> <p>This sanity check within <code>dao.protocol.setting.deposit</code> is missing within the <code>proposalSettingUint</code> function.</p> <p>Please note that this issue is only informational due to the <code>onlyValidSetting</code> modifier and the way how the settings were determined within the constructor of the <code>ProtocolSettingsSecurity</code> contract. If there will be more valid settings introduced in a future upgrade, it is advised to cross-check sanity checks and external calls.</p>
<b>Recommendations</b>	Consider keeping this in mind for future upgrades.
<b>Comments / Resolution</b>	<p>Acknowledged. Rocketpool considers this as a non-issue and added the following comment:</p> <p>“This is intentional. The security council may have different guardrails to the pDAO and so we will implement them as required if new settings are added to the security council’s authority. The settings the SC can modify are currently very limited and are settings that don’t have guardrails.”</p>

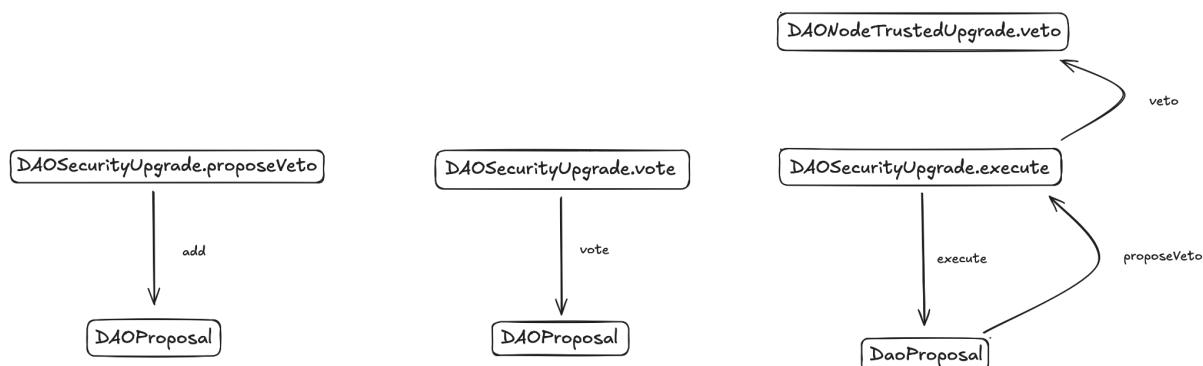
<b>Issue_15</b>	Removal of <code>rocketDAOProtocol</code> executions
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>Within resolution 1, the <code>onlyExecutingContracts</code> modifier was tightened to not prohibit the <code>rocketDAOProtocol</code> contract from calling corresponding functions.</p> <p>While we have identified that this is only related to the bootstrap mechanism, which is disabled, this is still a significant change which would be carefully evaluated, ensuring no side-effects in the standard business logic are introduced.</p> <p>This is especially concerning because it forms an inconsistency compared to the <code>RocketDAONodeTrustedProposals</code> contract.</p>
<b>Recommendations</b>	Consider being careful with such a change.
<b>Comments / Resolution</b>	Acknowledged.

## RocketDAO Security Upgrade

The [RocketDAOSecurityUpgrade](#) contract is a part of the security module but deployed as a standalone contract. It allows security members to propose a veto for ongoing upgrades that are created via the [RocketDAONodeTrustedUpgrade](#) contract. A veto is created in a similar fashion as a standard proposal and allows other security members to vote in this veto. This veto can then be executed via the [DAOProposals](#) contract, given it has sufficient `getUpgradeVetoQuorum`.

If that is successful, an upgrade can be permanently prevented

### Appendix: Veto Flow



As one can see, there are three different flows:

- initial proposal
- voting period
- veto execution

### Core Invariants:

INV 1: Only security members can call propose and veto

INV 2: Only the origin proposer can call cancel

INV 3: proposalVeto must only be called via DAOProposal contract

INV 4: A member cannot vote twice

## Privileged Functions

- proposalVeto

<b>Issue_16</b>	Unit mismatch in quorum logic
<b>Severity</b>	High
<b>Description</b>	<p>The <code>DAOProposal</code> contract determines a specific quorum which must be reached when the add function is called:</p> <pre>setUint[keccak256[abi.encodePacked[daoProposalNameSpace,     "votes.required", proposalID]], _votesRequired]; // How many votes are required for the proposal to pass</pre> <p>This is correctly handled within the <code>DAONodeTrustedProposals</code> and <code>DAOSecurityProposals</code>, as it uses the veto threshold and multiplies it with the member count.</p> <p>However, within the <code>RocketDAOUpgrade</code> contract, it uses the raw number:</p> <pre>rocketDAOProtocolSettingsSecurity.getUpgradeVetoQuorum()</pre> <p>which is set to 0.33 ETH. This crucial step is missing the incorporation of the total member count within the security module.</p>
<b>Recommendations</b>	Consider using the correct value.
<b>Comments / Resolution</b>	Resolved.

<b>Issue_17</b>	Removal of <code>rocketDAOProtocol</code> executions
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>Within resolution 1, the <code>onlyExecutingContracts</code> modifier was tightened to not prohibit the <code>rocketDAOProtocol</code> contract from calling corresponding functions.</p> <p>While we have identified that this is only related to the bootstrap mechanism, which is disabled, this is still a significant change which would be carefully evaluated, ensuring no side-effects in the standard business logic are introduced.</p> <p>This is especially concerning because it forms an inconsistency compared to the <code>RocketDAONodeTrustedProposals</code> contract.</p>
<b>Recommendations</b>	Consider being careful with such a change.
<b>Comments / Resolution</b>	Acknowledged.

## DAO/base

### RocketDAOProposal

The [RocketDAOProposal](#) contract is responsible for storing all created proposals and their current voting state. The following contracts can create proposals:

- DAONodeTrustedProposals
- DAOSecurityProposals
- DAOSecurityUpgrade

Each [proposalID](#) is unique and mapped to the corresponding DAO contract. The codebase implements a mechanism to handle all modules and at the same time ensures there are no collisions.

**This contract has not been changed for the Saturn upgrade.**

### Appendix: State Explanations

#### Cancelled

- This state is reached if the proposer self cancels

#### Executed

- This state is reached once the proposal succeeded and has been executed

#### Pending

- This is the initial state after proposal creation and before votes are allowed

#### Succeeded

- This state is reached when votes are allowed and the quorum has been reached but the proposal has not yet been executed

#### Active

- This state is reached after the Pending state, while the quorum has not yet been reached and before the end has reached

## Defeated

- This state is reached after end when the quorum has not been reached or  $\text{votesAgainst} \geq \text{votesFor}$

## Expired

- This state is reached when the proposal has satisfied success criteria at some point, but passes *expires* without execution.

Ordering note: The function checks Cancelled → Executed → Pending → Succeeded → Active → Defeated → Expired. This ordering makes an “early-success” proposal report Succeeded even if still before the end.

## Core Invariants:

INV 1: Only DAO contracts can call add/vote/cancel

INV 2: Only the correct corresponding DAO contract related to the proposalID can call vote/cancel

## Privileged Functions

- add
- vote
- cancel

# Deposit

## RocketDepositPool

The **RocketDepositPool** contract is the entry contract for users to mint RETH with ETH and handles the assignment of Minipools and Megapools via the existing ETH.

The contract has different mechanics which will be explained each in their own appendix:

**This contract has been changed for the Saturn upgrade.**

### Appendix: Deposit Process

Users can deposit native ETH and mint RETH for a small fee, which is by default 0.05% and can be set up to 1%. This is handled within the deposit function but only under the premise that the capacity (the user required amount for Megapool and Minipools) is sufficient to absorb the deposit, if the **maximumDepositPoolSize** is exceeded. If the **maximumDepositPoolSize** is not exceeded, users can freely deposit even if there is currently no capacity. This size is entitled by default with 160 ETH.

### Appendix: Assignments

Any deposited ETH from users will be used as the user part for Minipools and Megapools. For Minipools, the user part can be 24 ETH or 16 ETH and for Megapools, the user part is currently 28 ETH. The user part for Megapools may increase in future in the scenario where the required node bond is reduced.

After the upgrade, no more Minipools can be created, however, there will still be Minipools in the queue which are waiting to be assigned. Minipools have priority status before Megapools and Megapools will only be assigned after the queue for Minipools is empty.

For assignments, there are two specific ways:

#### a) Assignment during a new deposit/recycle:

Whenever the **processDeposit** function is triggered (new deposit/recycling), the **\_assignByDeposit** function is triggered which then assigns Minipools/Megapools based on the provided msg.value from the deposit/recycle operation (clamped by the maximum ETH amount in the pool).

## b] General assignment:

Whenever `maybeAssignDeposits` or `assignDeposits` is triggered, it will attempt to assign as many Minipools/Megapools as the `_max` parameter [clamped by the maximum ETH amount in the pool]. This is only possible if governance activated the `assignDepositsEnabled` state.

Assignments simply withdraw the required ETH amount from the RocketVault contract and then trigger the specific function on the corresponding pool type. For Minipools that is a simple `Minipool.deposit` call with 31 ETH attached and for Megapools that is a `Megapool.assignFunds` call with 32 ETH attached. Furthermore, the specific pool is removed from the queue. Minipools are using the `RocketMinipoolQueue` contract and Megapools the `LinkedListStorage` contract.

## Appendix: Recycling

The recycle process simply uses the provided `msg.value` and attempts to assign a Minipool/Megapool based on it/deposits the ETH into the RocketVault. A specific edge-case is handled whenever there is insufficient collateral within TokenRETH. In that scenario, it is attempted to decrease the shortfall to meet the target collateral ratio, which is 10% of the overall RETH supply.

Recycling happens whenever a Megapool or Minipool is dissolved or dequeued. It can also happen if the `TokenRETH` balance is above the target collateralization and will then simply transfer the excess into the `RocketVault` contract on behalf of the `DepositPool` contract.

Additionally, users are awarded credit for bond reductions which can be withdrawn from the deposit pool in the form of `TokenRETH`.

## Appendix: Express Queue

The contract implements a standard and express queue mechanism for the assignment of Megapools. Megapools in the express queue are executed with a rate of 2:1, which means the following sequence:

EXPRESS - EXPRESS - STANDARD /repeat



## Core Invariants:

INV 1: [RocketDepositPool.nodeDeposit](#) must always deposit bondAmount for Megapool creation and increase nodeBalanceKey likewise

INV 2: [\\_assignMinipoolsByDeposit](#)/Megapools must use the deposited msg.value as indicator for how much pools are assigned

INV 3: [Minipool/Megapool](#) assignment must decrease nodeBalanceKey by the node bond

INV 4: [recycleDissolvedDeposit](#) and [recycleExcessCollateral](#) must only happen with user share

INV 5: [requestedTotalKey](#) must keep track of requested funds for a megapool validator

INV 6: expressQueue must process with a rate of 2 over normal queue

INV 7: Dequeue of validator from Megapool is only allowed for validator that is in queue

INV 8: Each new validator for Megapool requires 4 ETH bond

INV 9: Funds for Minipool/Megapool assignments must be taken from RocketVault

INV 10: Upon [\\_assignMinipools](#), it must always deposit 31 ETH to the Minipool to achieve 32 ETH

INV 11: Any remaining [variableDeposit](#) Minipools in the queue must have the assigned prelaunchBalance of 1e18

INV 12: [\\_increaseETHBonded](#) must push a new snapshot

INV 13: RocketDepositPool contract must never hold any ETH

INV 14: During [\\_assignMinipoolsByDeposit](#), can never assign to more Minipools than getBalance / variableDepositAmount

INV 15: During [\\_assignByDeposit](#), it must only assign MegaPools if there are no Minipools in the queue

INV 16: Deposits must be  $\geq$  minimumDeposit



## Privileged Functions

- recycleDissolvedDeposit
- recycleExcessCollateral
- recycleLiquidatedStake [unused]
- withdrawExcessBalance
- exitQueue
- applyCredit

<b>Issue_18</b>	Incorrect behavior in <code>applyCredit</code> and <code>withdrawCredit</code> will skew RETH exchange ratio
<b>Severity</b>	High
<b>Description</b>	<p>The <code>applyCredit</code> and <code>withdrawCredit</code> function both mutate the <code>nodeBalanceKey</code> variable.</p> <p>This is incorrect as during credit application it will then double reduce the network eth balance and skews the ETH exchange rate.</p>
<b>Recommendations</b>	<p>The <code>applyCredit</code> and <code>withdrawCredit</code> function should not mutate <code>nodeBalanceKey</code>.</p> <p>There are also other issues related to credit accounting which need to be fixed. A whole re-audit of the adjusted control-flow is required to ensure accuracy of operations.</p>
<b>Comments / Resolution</b>	<p>Open. The overall credit mechanism has been refactored which includes a lot of changes to the <code>NodeDeposit</code> contract. While in fact changes to the <code>DepositPool</code> contract have been kept to a minimum, it is required to fully re-review the <code>NodeDeposit</code> contract and <code>DepositPool</code> contract and Megapool state transitions in contexts of all different control-flows. This will be the only possible way to fully confirm that fixes in itself work correctly and no edge-cases appear.</p> <p>This issue will be marked as resolved in the next iteration of the report.</p>
<b>Resolution 2 / Follow-Up</b>	<p>Resolved, the crediting mechanism has been refactored and it is now ensured that <code>nodeBalanceKey</code> and <code>node.deposit.credit.balance</code> transitions are always properly handled.</p> <p>This includes:</p> <ul style="list-style-type: none"> <li>- <code>reduceBond</code> control-flow</li> <li>- <code>dequeue</code> control flow</li> <li>- credit withdrawal control-flow</li> <li>- validator creation with credit control-flow</li> </ul> <p>Within the architecture, these are the only concepts that handle credit, as there are no further Minipools to promote which means there will not be any crediting mechanism via Minipools.</p>

Issue_19	Incorrect behavior during <code>dequeue</code> control-flow will result in skewed RETH exchange rate
Severity	High
Description	<p>A Megapool which is currently in the queue phase has already accounted for <code>userCapital</code> and <code>nodeBond</code>.</p> <p>This is happening during <code>newValidator</code>. It is important to mention that within the queued state, this validator does not yet fall into <code>network.eth.balance</code>, because the corresponding 32 ETH are still sitting in the <code>UserDepositBalance</code>.</p> <p>This is properly handled via various different mechanisms.</p> <p>An issue/exception here is the <code>dequeue</code> control-flow, as it calls <code>applyCredit</code> which in turn increases <code>node.deposit.credit.balance</code>. The problem with this is that the increase of the said credit balance now has an inherent deduction impact on <code>network.eth.balance</code> while the megapool itself never had any impact just yet.</p> <p>It will only result in a <code>userDepositBalance</code> of 28 (we use bond = 4) in case the megapool is actually assigned/staked, but not yet when it is in the queue.</p> <p>The root case is that within <code>dequeue</code>, it will increase <code>node.deposit.credit.balance</code> while the actual Megapool did not have any impact during its queued state on the total eth balance.</p>
Recommendations	To fix this issue, it is required to refactor the whole dequeue flow. A full re-audit of the credit logic is mandatory to ensure accuracy.
Comments / Resolution	Open. The overall credit mechanism has been refactored which includes a lot of changes to the NodeDeposit contract. While in fact changes to the DepositPool contract have been kept to a minimum, it is required to fully re-review the NodeDeposit contract and DepositPool contract and Megapool state transitions in contexts of all different control-flows. This will be the only possible way to fully confirm that fixes in itself work correctly and no edge-cases

	<p>appear.</p> <p>This issue will be marked as resolved in the next iteration of the report.</p>
<b>Resolution 2 / Follow-Up</b>	<p>Resolved, the crediting mechanism has been refactored and it is now ensured that <code>nodeBalanceKey</code> and <code>node.deposit.credit.balance</code> transitions are always properly handled.</p> <p>This includes:</p> <ul style="list-style-type: none"><li>- <code>reduceBond</code> control-flow</li><li>- <code>dequeue</code> control flow</li><li>- credit withdrawal control-flow</li><li>- validator creation with credit control-flow</li></ul> <p>Within the architecture, these are the only concepts that handle credit, as there are no further Minipools to promote which means there will not be any crediting mechanism via Minipools.</p>

Issue_20	Incorrect behavior during validator creation with credit
Severity	High
Description	<p>Whenever a new validator for a megapool is created with credit, it follows a very specific control-flow. Specifically, it will call <code>nodeDeposit</code> which increases <code>nodeBalanceKey</code> by the provided <code>msg.value</code>:</p> <pre><code>addUint(nodeBalanceKey, msg.value);</code></pre> <p>This behavior is incorrect as in the scenario of <code>msg.value &lt; bondAmount</code> [use <code>msg.value = 0</code> for simplification], it will not increase <code>nodeBalanceKey</code> while the actual credit pointer is decreased. The effect is that the creation of a validator via this control-flow will skew the exchange rate and incorrectly increase the total network balance.</p> <p>Illustrated, you can consider the following scenario:</p> <ul style="list-style-type: none"> <li>a] User has 32 ETH validator. It is not vacant at this moment. No impact on exchange rate</li> <li>-</li> <li>b] User calls <code>prepareVacancy</code>, <code>nodeDepositBalance = 8 ETH</code>; <code>userDepositBalance = 24 ETH</code>. No impact at this point</li> <li>c] No scrub is happening, promote is called.</li> </ul> <p>&gt; <code>userDepositBalance = 24 ETH</code> flows into exchange rate [ER]; increases ER  &gt; increase credit; decreases ER  &gt; at this point &gt; neutral</p> <p>the user has now credit + <code>userDepositBalance</code> of the promoted minipool flows into the ER. The scenario is perfectly handled as there is no impact on the exchange rate</p> <ul style="list-style-type: none"> <li>d] User creates validator with credit  &gt; decrease credit: increase ER</li> </ul>

	<p>&gt; call <code>nodeDeposit</code> with <code>msg.value = 0</code></p> <p>&gt; nothing happens; ER remains elevated</p> <p>&gt; it is pushed into the queue but at this point the storage of Megapool is not considered for exchange rate</p> <p>&gt; In that scenario, the decrease of the credit should actually be countered by an increase of <code>nodeBalanceKey</code> to make sure that this stays fully neutral</p> <p>The exact same issue happens if we get credit on any other scenario.</p> <p>The root-cause stems from the <code>nodeDeposit</code> function which only increases <code>nodeBalanceKey</code> by <code>msg.value</code> which excludes the credit. The assumption that the credit is already included in <code>nodeBalanceKey</code> is incorrect.</p>
<b>Recommendations</b>	<p>Consider increasing <code>nodeBalanceKey</code> by the actual node bond. This removes the assumption that any credit has already been included in <code>nodeBalanceKey</code> (the NATSPEC is incorrect).</p> <p>Since there are multiple issues related to the credit accounting, a full re-audit of all control-flows is required after fixes have been applied.</p>
<b>Comments / Resolution</b>	<p>Open. The overall credit mechanism has been refactored which includes a lot of changes to the <code>NodeDeposit</code> contract. While in fact changes to the <code>DepositPool</code> contract have been kept to a minimum, it is required to fully re-review the <code>NodeDeposit</code> contract and <code>DepositPool</code> contract and Megapool state transitions in contexts of all different control-flows. This will be the only possible way to fully confirm that fixes in itself work correctly and no edge-cases appear.</p> <p>This issue will be marked as resolved in the next iteration of the report.</p>
<b>Resolution 2/ Follow-Up</b>	Resolved, the crediting mechanism has been refactored and it is now ensured that <code>nodeBalanceKey</code> and <code>node.deposit.credit.balance</code> transitions are always properly handled.

This includes:

- `reduceBond` control-flow
- `dequeue` control flow
- `credit withdrawal` control-flow
- `validator creation with credit` control-flow

Within the architecture, these are the only concepts that handle credit, as there are no further Minipools to promote which means there will not be any crediting mechanism via Minipools.

<b>Issue_21</b>	Crediting fees can be bypassed in case of no dissolve penalty [FOLLOWUP]
<b>Severity</b>	Medium
<b>Description</b>	<p>It is possible that there is no dissolve penalty. In such a scenario, crediting fees can be bypassed in an edge-case:</p> <ul style="list-style-type: none"> <li>- Alice has two validators and 1 ETH credit</li> <li>- Governance decreases reduced.bond to 2 ETH</li> <li>- Alice queues validator 3 with own withdrawal credentials: <ul style="list-style-type: none"> <li>- requiredBondAmount = [4,2,2]</li> <li>- effectiveBond = 8</li> <li>- due to the maxing of prestateValue, Alice needs to provide 1 ETH as bond</li> </ul> </li> <li>- The 1 ETH is given directly back to Alice's withdrawal address [if she funded 31 more ETH directly and exited]</li> <li>- Alice now dissolves validator 3 without any penalty [it can be zero] <ul style="list-style-type: none"> <li>- effectiveBond = 9</li> <li>- requiredBondAmount = [4,2]</li> <li>- _nodeShare = 3</li> <li>- _userShare = 29</li> <li>- toNode = 2</li> <li>- Alice gets 2 ETH back</li> </ul> </li> <li>- Alice now receives a total of 2 ETH [from bond reduction] and 1ETH [from credit] and saves on the fees</li> </ul>
<b>Recommendations</b>	Consider never allowing for a dissolve fee of zero.
<b>Comments / Resolution</b>	Resolved.

Issue_22	Lack of slippage during <code>mint</code> in case of fee change race condition
Severity	Medium
Description	The <code>deposit</code> function takes a fee for RETH minting. Due to the fact that the fee is a dynamic value, it can happen that this value increased before a transaction was executed. This will then result in a user paying more fee than expected.
Recommendations	Consider implementing a slippage check.
Comments / Resolution	Acknowledged.

Issue_23	Legacy credit holders cannot withdraw credit without megapool creation [FOLLOWUP]
Severity	Low
Description	<p>The <code>_withdrawCreditFor()</code> function enforces a hard requirement that the node operator must have a deployed megapool:</p> <pre><code>require[rocketMegapoolFactory.getMegapoolDeployed[_nodeAddress], "Megapool must be deployed"];</code></pre> <p>However, legacy minipool operators can accumulate credit via <code>increaseDepositCreditBalance()</code> when their minipools dissolve - this function is explicitly restricted to minipools only.</p> <p>This creates a situation where:</p> <ol style="list-style-type: none"> <li>1. Legacy node operator runs minipools (no megapool)</li> <li>2. Minipool dissolves, credit is issued to the node</li> <li>3. Node attempts to withdraw credit as rETH</li> <li>4. Transaction reverts with "Megapool must be deployed"</li> </ol> <p>The node operator is forced to deploy a megapool (incurring gas costs and protocol commitment) solely to access credit they legitimately earned from legacy minipool operations. This is particularly problematic for operators wishing to exit the protocol entirely - they cannot retrieve their credit without first deploying new infrastructure.</p>
Recommendations	<p>Consider whether it makes sense to add an alternative path to withdraw credit. However, that would introduce significant challenges in terms of user flexibility and is very prone to new issues.</p> <p>Therefore, incorporating the fact that it is not too likely to get credits from dissolved minipools, we recommend acknowledging this issue.</p>
Comments / Resolution	Acknowledged.

<b>Issue_24</b>	Lack of shortfall incorporation into capacity will prevent reaching target collateralization
<b>Severity</b>	<b>Low</b>
<b>Description</b>	<p>Within <code>RocketDepositPool.deposit</code>, it does not include potential shortfall of RETH into capacity. This can result in longer undercollateralization as expected because essentially deposits are blocked.</p> <p>The <code>deposit</code> function enforces capacityNeeded as:</p> $\text{capacityNeeded} = \text{getBalance}[] + \text{msg.value};$ <p>This treats all of <code>msg.value</code> as if it will increase the deposit pool balance, but a portion of <code>msg.value</code> (up to the rETH collateral shortfall) is immediately forwarded to the rETH contract in the same transaction by <code>processDeposit()</code> and does not end up in the deposit pool.</p> <p>This means that the capacity limit might not be exhausted which is a problem if the queue is empty.</p> <p>Consider a case where there is 160 ETH in the pool while there are no outstanding queues, essentially no <code>processDeposit</code> will be queued and it's impossible to collateralize RETH via the <code>processDeposit</code> control-flow.</p> <p>It should ideally first calculate the shortfall and then add this on top of the capacity</p> <p>Marked this as low because in this specific scenario, RETH redemptions will still work because they transfer ETH -&gt; RETH to honor the withdrawal.</p> <p>The only issue is that there is no automatic/manual target collateralization here</p>
<b>Recommendations</b>	A solution for this issue is to incorporate potential shortfall into the

	capacity logic. However, since that change alters the control-flow and that issue is not deemed as severe, we recommend acknowledging it.
Comments / Resolution	Acknowledged.

Issue_25	Inconsistency within <code>getQueueTop</code> and <code>_assignMegapools</code> express determination logic
Severity	Low
Description	<p>Within the <code>getQueueTop</code> and <code>_assignMegapools</code> function, there is an inconsistency in how the express logic is determined, specifically, within <code>getQueueTop</code> it is determined as follows:</p> <pre>bool express = queueIndex % (expressQueueRate + 1) != 0;</pre> <p>That means essentially, the very first index is not treated as express, which is false.</p>
Recommendations	Consider following the determination logic within <code>_assignMegapools</code> .
Comments / Resolution	Resolved.

<b>Issue_26</b>	Revert due to non dynamic deposit amount adjustment in case deposit is above maxDepositPoolSize
<b>Severity</b>	Low
<b>Description</b>	<p>The <code>deposit</code> function exposes a minimum deposit but also a <code>maxDepositPoolSize</code>.</p> <p>If the leftover size is smaller than the minimum deposit, it is impossible to fully fill the pool.</p>
<b>Recommendations</b>	Consider acknowledging this issue. While it would be theoretically possible to introduce an extra control-flow for this scenario which clamps the minimum deposit size, this will be an intrusive change which we don't recommend.
<b>Comments / Resolution</b>	Acknowledged.

<b>Issue_27</b>	Lack of automatic Megapool assignment in case Minipool queue became empty
<b>Severity</b>	Low
<b>Description</b>	Currently, the queue assignment favors minipoools and only assigns megapoools once all minipoools have been dequeued. In the scenario where an assignment has X capacity but there are only X - 5 minipoools, it does not automatically continue to assign 5 megapoools and simply stops with the minipoools.
<b>Recommendations</b>	Since this is only a small performance issue and no security threat and will only occur once because no new minipoools can be created anymore, we recommend safely acknowledging it.
<b>Comments / Resolution</b>	Acknowledged.

<b>Issue_28</b>	Inconsistency in <code>getUserBalance</code> between <code>dequeue</code> and <code>reduceBond</code> [FOLLOWUP]
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>The <code>dequeue</code> control-flow will decrease <code>nodeBalanceKey</code> which influences the <code>getUserBalance</code> function, while the <code>reduceBond</code> function will increase the credit which does not influence the <code>getUserBalance</code> function.</p> <p>In terms of exchange rates, both functions are correct because the <code>reduceBond</code> function will increase <code>userCapital</code>.</p>
<b>Recommendations</b>	Consider acknowledging this issue.
<b>Comments / Resolution</b>	Acknowledged.

<b>Issue_29</b>	Incorrect casting to <code>uint64</code> within <code>getQueueTop</code>
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>Within the <code>getQueueTop</code> function, <code>packed</code> is casted as <code>uint64</code>:</p> <pre>if [express] {     headMovedBlock = uint64[packed]; } else {     headMovedBlock = uint64[packed &gt;&gt; 128]; }</pre> <p>However, this value is actually a <code>uint128</code>.</p>
<b>Recommendations</b>	Consider correcting the casting.
<b>Comments / Resolution</b>	Resolved.

<b>Issue_30</b>	Incorrect comment for withdrawalAddress [FOLLOWUP]
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>The following comment is incorrect:</p> <p><i>// Get the node operator's withdrawal address</i></p> <p>Such a behavior does not exist in the current code.</p>
<b>Recommendations</b>	Consider removing that comment.
<b>Comments / Resolution</b>	Resolved.

# Megapool

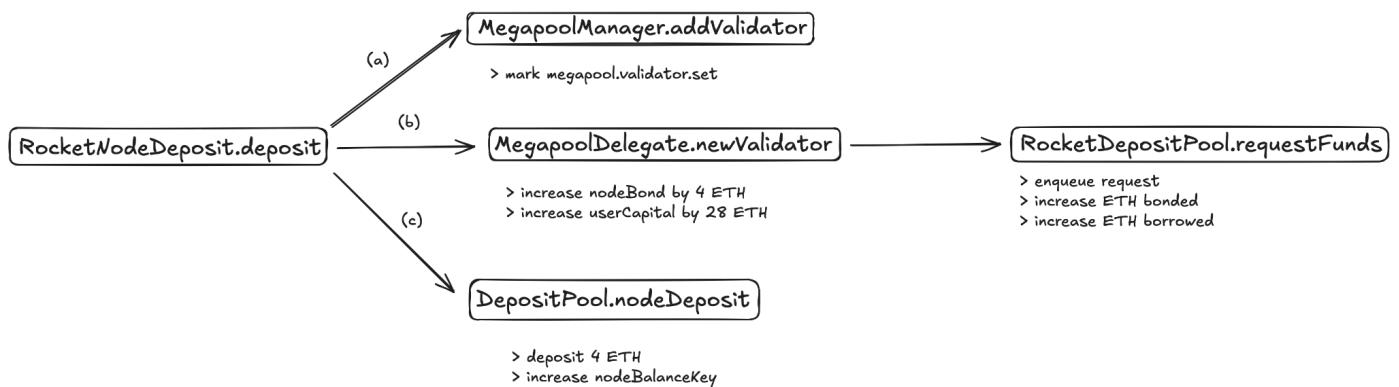
## RocketMegapoolDelegate

The [RocketMegapoolDelegate](#) contract is the heart of the Megapool module and exposes the implementation contract for Megapools. A modular approach has been chosen to ensure that individual Megapools can be upgraded and the [MegapoolProxy](#) contract then simply points to a new [RocketMegapoolDelegate](#) contract.

The Megapool has different mechanisms and concepts which will be explained in the corresponding appendices.

### Appendix: Validator Creation

Any registered node can create a validator simply by executing a deposit call on the [NodeDeposit](#) contract, the flow for adding a new validator and pushing it into the queue is as follows:

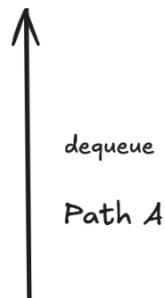


### Appendix: Validator Lifecycle

In the above appendix, we already explained how a new validator for a megapool is created, which is simply through the [DepositPool](#) contract and a queue mechanism. It is important to mention that the creation is already part of the lifecycle. Below we will illustrate the different parts of the lifecycle:

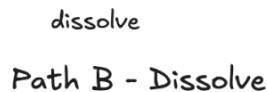
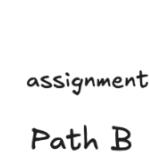
## Dequeued Validator

- > userCapital is decreased
- > Megapool is removed from the queue
- > requestedTotalKey is decreased
- > decrease bonded/borrowed ETH
- > nodeBond is decreased
- > credit is granted
- > numActiveValidators is increased



## Enqueued Validator

- > validator has been created
- > Megapool is sitting in the queue

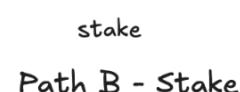


## Assigned Validator

- > item is removed from the queue
- > validator is assigned
- > ValidatorInfo is adjusted
- > assignedValue is increased
- > 1 ETH is prestaked
- > 31 ETH are sitting in Megapool

## Dissolved Validator

- > adjust ValidatorInfo
- > nodeBond/userCapital is decreased
- > numInactiveValidators is increased
- > assignedValue is decreased
- > recycle user share
- > increase refund value
- > decrease bonded/borrowed ETH



## Staked Validator

- > decrease assignedValue
- > adjust ValidatorInfo

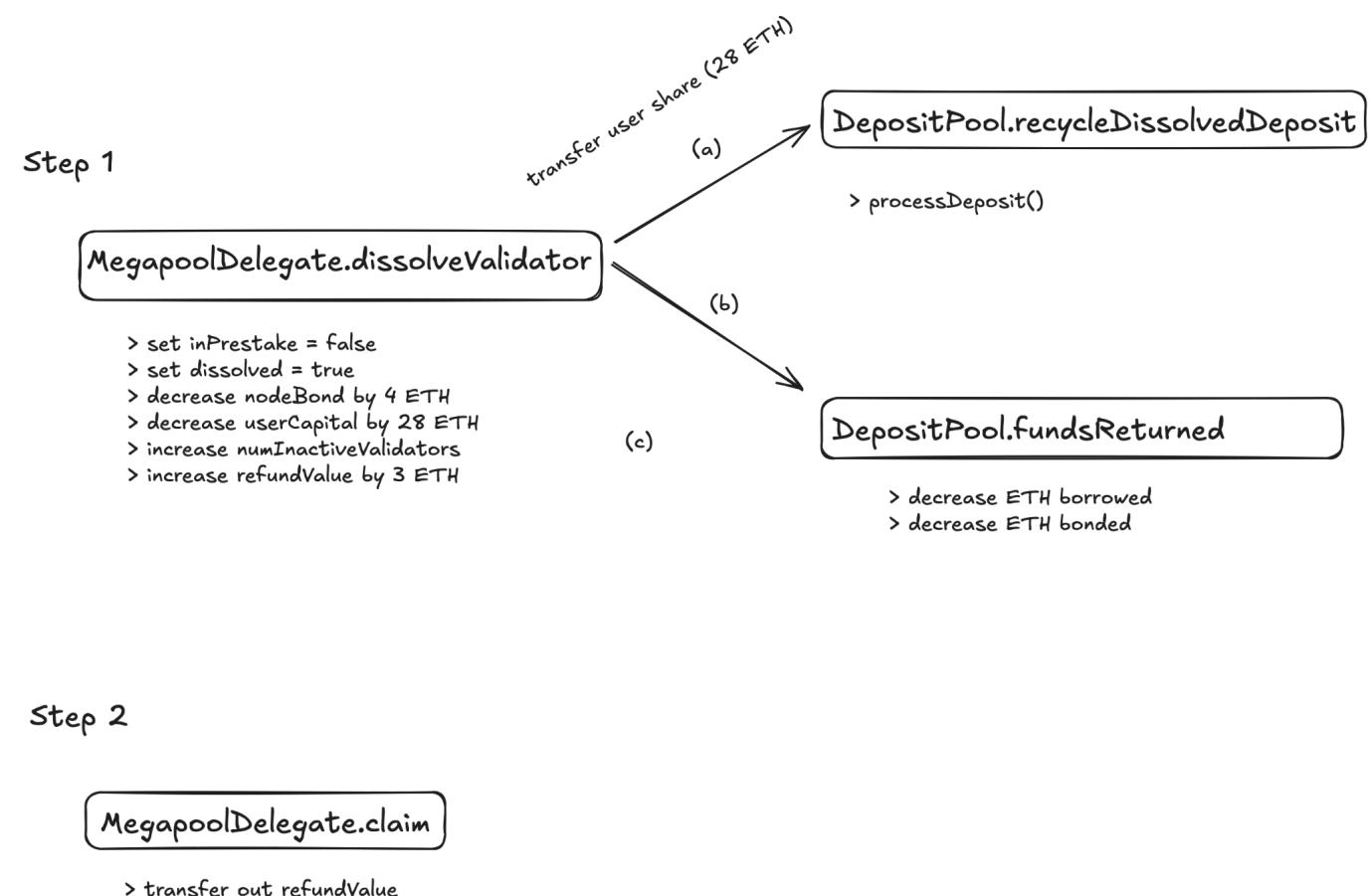
## Appendix: Validator Dissolve

A validator can be dissolved whenever it **has been assigned and is not yet staked**. This can happen in two scenarios:

- a) The validator does not have the correct withdrawal credentials
- b) The `stake` function has not been called within a period of 28 days

The goal of the dissolve process is to make sure that the `userCapital` as well as the provided `nodeBond` is redistributed back to the DepositPool

The flow for this process is as follows:



It is important to mention that in the current code pattern, a dissolved validator remains in the dissolved state and will never get moved into the exiting nor exited state. This has the background that it is expected that dissolving only happens if the credentials do not match or when staking has not been completed within 28 days.

## Appendix: Validator Exit

Whenever a validator is exiting on the Beacon Chain (see Appendix Beacon Chain: Withdrawals), it is required to notify the Megapool of this exit. This is done as follows:

- a] `Megapool.notifyExit`
- b] Wait until `withdrawable_epoch` is reached AND withdrawal is executed
- c] `Megapool.notifyFinalBalance`

## Appendix: Debt

Debt is the result of slashing for a validator which will be globally reflected on the Megapool and is not isolated to each validator. A validator gets slashed on the following occasions:

- a] Withdrawal with shortfall
- b] Governance vote

This then increases the debt parameter on the Megapool and as long as a Megapool has non-zero debt, it is prohibited to add a new validator. Debt can be repaid either via

- Direct call to `repayDebt` with native ETH attachment
- Decrease of rewards during `distribute`
- Decrease of `nodeBond` during exit

## Appendix: Capital Dispersal

Whenever rewards are distributed within the `distribute` function, these are calculated based on the overall rewards

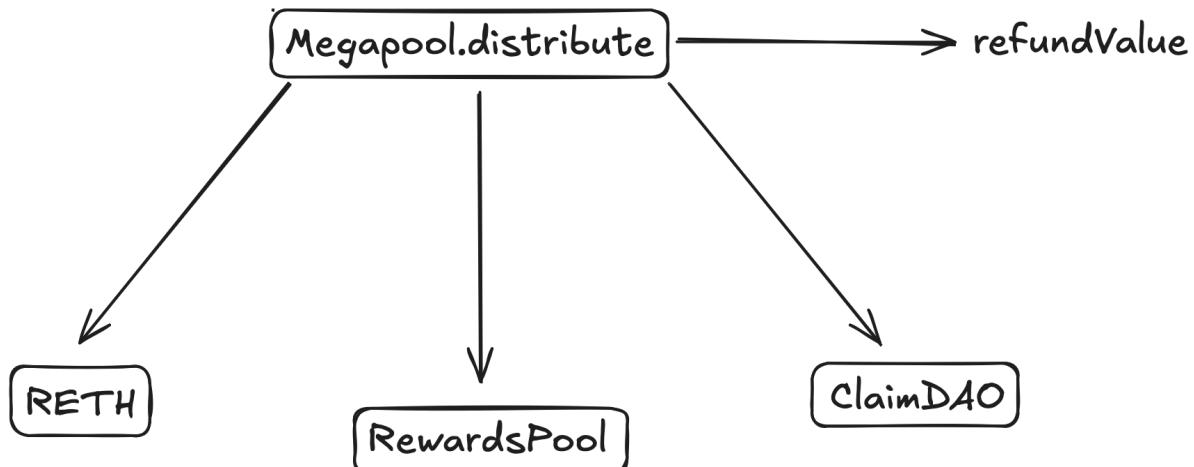
> `[address[this.balance - refundValue - assignedValue]]`

and the following formula:

1. Calculate borrowedPortion: `rewards * userCapital / [nodeBond + userCapital]`
2. Calculate rethShare: `rethShare * borrowedPortion / 1e18`
3. Calculate voterShare: `voterShare * borrowedPortion / 1e18`
4. Calculate protocolDAORewards: `protocolDAOShare * borrowedPortion / 1e18`
5. Calculate nodeRewards: `rewards - rethShare - voterShare - protocolDAORewards`

Notable, the `nodeShare` (`network.node.commission.share`) is applied on the borrowed amount and thus reflects a node's commission on the borrowed amount. The full leftover is additionally added to the `nodeShare`.

These are then distributed as follows:



#### Core Invariants:

INV 1: During second deposit of Megapool, [depositDataRoot](#) is computed based on dummy signature

INV 2: [Megapool.dequeue](#) is only allowed if the validator has not been assigned yet.

INV 3: Cannot distribute a Megapool with [existing validators](#)

INV 4: Cannot distribute a Megapool which [is challenged](#)

INV 5: During `notifyNotExit`, proof must [not be older than challenged slot](#)

INV 6: During [assignFunds](#), must provide 32 ETH to the Megapool (node/user share)

INV 7: Megapool.debt can only be reduced during [direct repay](#), [reward distribution/claiming](#) or [notifyFinalBalance](#)

INV 8: validator can only dissolved after assignment and before stake

INV 9: During [dissolveValidator](#), nodeShare must remain in Megapool

INV 10: Before dissolveValidator, balance must be minimum 31 ETH

INV 11: `_distributeAmount` must be called before [`reduceBond`](#)

INV 12: sum of userCapital and nodeAmount must always be 32 ETH batches

INV 13: refundValue must be distributed to nodeWithdrawalAddress

INV 14: `_distributeAmount` is not allowed if soonestDrawableEpoch has been reached

INV 15: A new validator for Megapool can only be added if there is [`no debt`](#)

INV 16: [`assignFunds`](#) must be called on every Megapool assignment

INV 17: Whenever a Megapool is assigned, it must be set to inPrestake

INV 18: During Megapool assignment, it must deposit 32 ETH into the Megapool and 1 ETH from the Megapool into the ETH deposit contract

INV 19: After Megapool assignment, anyone can call [`Megapool.stake`](#) to stake the remaining 31 ETH

INV 20: ValidatorID from Megapool starts with 1

INV 21: Validators always have a 0x01 withdrawal address

## Privileged Functions

- `newValidator`
- `assignFunds`
- `stake`
- `notifyExit`
- `notifyFinalBalance`
- `challengeExit`
- `notifyNotExit`
- `applyPenalty`

Issue_31	<p>dissolveValidator + notifyFinalBalance control-flow can be exploited to inflate refundValue maliciously [FOLLOWUP]</p>
Severity	High
Description	<p>Megapool lifecycle includes:</p> <ul style="list-style-type: none"> <li>- <code>assignFunds</code>: sets <code>validator.inPrestake = true</code> and performs a 1 ETH deposit to the beacon deposit contract.</li> <li>- <code>stake</code>: later deposits the remaining ETH [31] after a state proof that the validator is not activated/exiting.</li> <li>- <code>notifyExit</code> and <code>notifyFinalBalance</code>: state-proof-based lifecycle transitions for exiting and final withdrawal settlement.</li> <li>- <code>challengeExit</code>: trusted-node mechanism that locks distribution for staked validators to prevent withdrawal funds being treated as rewards before exit is processed.</li> <li>- <code>notifyExit</code> is only allowed in the delegate when <code>validator.staked    validator.dissolved</code>.</li> <li>- <code>challengeExit</code> requires <code>validator.staked</code>.</li> <li>- dissolve validators can be included within <code>notifyFinalBalance</code></li> <li>- <code>notifyExit/notifyFinalBalance</code> does not enforce withdrawal credentials to match the megapool</li> </ul> <p>In <code>RocketMegapoolDelegate._notifyFinalBalance</code>, if <code>validator.dissolved == true</code>, the code takes a special path that simply does:</p> <pre style="font-family: monospace; margin-left: 20px;"><code>refundValue += withdrawalBalance;</code></pre> <p>[without any further accounting or verification that the ETH is present in the contract].</p> <p>This specific root-cause can be exploited via various different control flows by inflating <code>refundValue</code> to then steal subsequent idle ETH (after assignment and pre stake and bypassing <code>getPendingReward</code> underflow concerns).</p> <p><b>Example 1 (simple):</b></p> <ul style="list-style-type: none"> <li>- queue validator 1</li> </ul>

- direct deposit to validator 1 and set own credentials, can deposit a huge amount, e.g. 1000 ETH
- assign validator 1
- dissolve validator 1
- exit validator 1 on BC, receive all 1000 ETH back
- notifyExit validator 1 with real proof
- notifyFinalBalance validator 1 with real proof
- increase refundValue by 1000 ETH

**Example 2 [complex]:**

- deposit 32 ETH as usual to a validator on the BC and thus activate it, set withdrawal address to the Megapool we use in the next step. The NO spends 32 ETH for this technique.
  - NO = - 32
- request a withdrawal t0 = 0 for this validator on the BC. This request happens after validator was sufficient time active [config.SHARD\_COMMITTEE\_PERIOD]
- queue this validator for Megapool. The NO spends 4 ETH for this technique
  - NO = - 36
- assign this validator at t1 = 26.9h
- deposits another 1 ETH on this validator; its a simple top-up because the deposit happens after it is exiting
- will be redistributed after withdrawal has happened
- Validator is now in prestake mode
  - Note that there is absolutely no possibility at this time for anyone to call notifyExit [needs to be staked or dissolved] or challengeExit [needs to be staked] for this validator that has actually already exited and is expected to receive withdrawal shortly

- At  $t_2 = 27h$ : 32 ETH are now skimmed and distributed as reward (which is actually the full withdrawal), while the validator is in prestate mode (withdrawal has been included in execution payload of withdrawals list)
  - NO = -32
  - users = 28
  - Important note: We ignore the reward distribution pattern here and simply assign 4/28 for simplicity reasons
  - Important note 2: In case where there is no staked validator yet, [it will route all 32 ETH to the NO because lastDistributionTime = 0](#), but claim is guarded so there's no trivial exploit path to reach this. It would even work to not use the distribution step and directly call dissolveValidator because then it would enter exactly this state and the full 32 ETH is allocated to refundValue
- the validator is now dissolved in the megapool because he was already exited
  - NO = -29
  - users = 28
- Since the dissolved validator has a valid withdrawal proof, due to the exit at  $t_2 = 27h$ , he can call notifyExit and notifyFinalBalance within the dissolved validator state, entering the first if-condition
  - refundValue = 32 ETH
  - NO = +3
  - Important note: This step would revert due to underflow at the distribution step. Therefore, attacker can simply call challengeExit (if trusted node) before notifyFinalBalance is being called OR the attacker exits another validator, to increase numLockedValidators or numExitedValidators. This step is easy to include in the exploit path

- Now a new validator is queued and assigned which sets ETH balance = 31 and refundValue can simply be withdrawn via claim
  - Important note: The queue would revert due to current debt increase during dissolveValidator. Not a problem because we can simply repay the debt. We can also pre-queue more validators before dissolving our malicious validator which bypasses this step.
  - Important note 2: The assignment only keeps 31 ETH because of 1 ETH being transferred to the deposit contract. This is not a blocker because we can simply assign more

As one can see from the NO value assignment, the attacker profited 3 ETH from it. This is very simplified and depends on the actual reward distribution ratio and should include dissolve penalty etc. Its simply to show that its possible, at the end it doesn't really matter whether its 1.5 or 3 ETH as long as this control-flow is allowed someone can repetitively steal from the protocol.

### Example 3 [medium complexity]:

- activate validator 1 on BC
- exit validator 1 on BC
- queue validator 1 on RP
- assign validator 1 on RP
- withdrawal arrives
  - dissolve validator 1; get 32 ETH as rewards  
[lastDistributionTime = 0, giving full 32 ETH to NO]
- queue validator 2
- assign validator 2
- stake validator 2
- exit validator 2 via notifyExit [numExitingValidators = 1]
  - 32 ETH (or more) from validator 2 is arriving
- finalize validator 1 via notifyExit and notifyFinalBalance with valid proof -> get 32 ETH as refundValue. The 32 ETH will be auto-claimed b/c the caller is the owner and the 32 ETH from

	<p>validator 2 are sitting now in the balance and will not be distributed as rewards [due to numExitingValidators = 1]</p> <p>Note that in example 2 and 3 it is even possible to steal funds from the protocol while the credentials from the dissolved validator match. Therefore it is not sufficient to only include a credential check during notifyExit</p>
<b>Recommendations</b>	Consider fully removing the control-flow of <code>notifyExit/notifyFinalBalance</code> for dissolved validators and implement a credential check within <code>notifyExit/notifyFinalBalance</code> .
<b>Comments / Resolution</b>	Resolved.

Issue_32	Permanently broken megapool and locked funds due to ETH donation after first <code>dissolveValidator</code> call [FOLLOWUP]
Severity	High
Description	<p><code>RocketMegapoolDelegate</code> distributes ETH held by the Megapool using <code>_distributeAmount(getPendingRewards[])</code>. This is invoked indirectly by <code>_snapshotCapitalRatio()</code>, which is called by major lifecycle transitions such as <code>stake</code> and <code>dissolveValidator</code>.</p> <p>Reward splitting relies on <code>RocketNetworkRevenues.getNodeAverageCapitalRatioSince(nodeAddress, lastDistributionTime)</code>, which requires that a <code>node.capital.ratio</code> snapshot exists.</p> <p>The Megapool tracks whether it has started distribution using <code>lastDistributionTime</code>. A “no first validator” guard exists in <code>distribute()</code> via <code>require(lastDistributionTime != 0, "No first validator")</code>, intended to prevent distributions before any validator has been staked.</p> <p><code>dissolveValidator</code> can call <code>_snapshotCapitalRatio()</code> in a state where <code>getPendingRewards() == 0</code> and <code>nodeBond == 0</code>. In that case:</p> <ul style="list-style-type: none"> <li>- <code>_snapshotCapitalRatio()</code> calls <code>_distributeAmount[0]</code>, which still sets <code>lastDistributionTime = block.timestamp</code>, even though no validator has ever been staked.</li> <li>- Because <code>nodeBond == 0</code>, <code>_snapshotCapitalRatio()</code> does not call <code>RocketNetworkRevenues.setNodeCapitalRatio</code>, so the node capital ratio snapshot is never created.</li> </ul> <p>After this state is reached, if any ETH is later forced into the Megapool (even 1 wei) via a donation attack, <code>getPendingRewards()</code> becomes non-zero. On the next call to <code>_snapshotCapitalRatio()</code> (e.g., during <code>stake</code> of a newly assigned validator), <code>_distributeAmount</code> calls <code>calculateRewards</code>, which calls <code>RocketNetworkRevenues.getNodeAverageCapitalRatioSince(...)</code> with <code>_mustExist = true</code>. Since the node capital ratio snapshot does not exist, the call reverts with "Snapshot does not exist".</p> <p>Because <code>_snapshotCapitalRatio</code> attempts reward distribution</p>

	<p><b>before</b> writing the capital ratio snapshot, the system enters a circular dependency and cannot recover.</p> <p>The result of this is that the Megapool is permanently locked while <code>assignFunds</code> is still possible which means funds remain trapped in the system.</p>
<b>Recommendations</b>	<p>Consider ensuring that there is always a valid <code>capitalRatio</code> in this specific scenario. This can be solved by a minimal call to <code>rocketNetworkRevenues.setNodeCapitalRatio</code> <b>after</b> <code>nodeBond/userCapital</code> are updated and only when <code>nodeBond &gt; 0</code>. Ideally, this also only happens for the very first assignment to ensure subsequent assignments do not alter the <code>capitalRatio</code> without having distributed any rewards.</p> <p>The idea of this practice is to solely eliminate the root-cause of the issue without altering any control-flow.</p> <p>This seems to be the least intrusive fix without calling <code>_snapshotCapitalRatio</code> during <code>assignFunds</code> - as this will introduce many edge-cases.</p> <p>However, it has to be ensured that such a setting does not result in critical side-effects.</p>
<b>Comments / Resolution</b>	Resolved.

Issue_33	<p>pendingRewards underflow due to manipulated refundValue can break megapool [FOLLOWUP]</p>
Severity	High
Description	<p>In the another we demonstrated how it is possible that <code>refundValue</code> is inflated without the corresponding ETH remaining in the contract balance, and then:</p> <ul style="list-style-type: none"> <li>- <code>address[this].balance - refundValue - assignedValue</code> underflows and reverts.</li> <li>- This revert can occur in <code>_snapshotCapitalRatio</code> and therefore bricks major state transitions.</li> </ul> <p>The code contains <b>no invariant enforcement</b> that <code>refundValue + assignedValue &lt;= address[this].balance</code> before increasing <code>refundValue</code> or before invoking <code>_snapshotCapitalRatio()</code>.</p> <p>Example:</p> <ul style="list-style-type: none"> <li>- activate validator 1 on BC</li> <li>- exit validator 1 on BC</li> <li>- queue validator 1 on RP</li> <li>- assign validator 1 on RP</li> <li>- withdrawal arrives dissolve validator 1; get 32 ETH as rewards (lastDistributionTime = 0, giving full 32 ETH to NO)</li> <li>- debt is repaid</li> <li>- notifyExit for validator 1</li> <li>- queue/assign/stake validator 2</li> <li>- challengeExit for validator 2 (trusted node)</li> <li>- notifyFinalBalance with valid proof -&gt; get 32 ETH as refundValue. The finalization must happen with an external address to bypass the auto claim path. The state after finalization will then permanently revert on each <code>_snapshotCapitalRatio</code> call, effectively locking the Megapool</li> <li>- notifyNotExit for validator 2</li> </ul>

	<p>It has to be noted that it will now always be possible to queue and assign more validators to the Megapool while these cannot be dissolved or staked due to the revert occasion. One can indefinitely lock funds from the DepositPool - however, this requires for the attacker to lock 4 ETH as well, while always being able to dequeue and regain funds. It can essentially be used to blackmail the protocol.</p>
<b>Recommendations</b>	Consider fully removing the control-flow of <code>notifyExit/notifyFinalBalance</code> for dissolved validators and implement a credential check within <code>notifyExit/notifyFinalBalance</code> .
<b>Comments / Resolution</b>	Resolved.

Issue_34	Increase of <code>reduced.bond</code> will break multiple transitions [FOLLOWUP]
Severity	High
Description	<p>Currently, it is possible for <code>reduced.bond</code> to be increased. This seems to be a potential edge-case which is meant to be handled properly.</p> <p>However, there are multiple instances where the code would break in such a practice:</p> <p>a) <code>dequeue</code> will revert if a megapool is underbonded after validators -1:</p> <pre>require[nodeBond + nodeQueuedBond &gt;= newBondRequirement, "Bond requirement not met"];</pre> <p>This can easily happen if a NO has many validators with 2 ETH bond requirement each and now the requirement is bumped up to e.g. 4 ETH</p> <p>b) megapool rewards are based on the ratio of <code>nodeBond</code>. If the <code>nodeBond</code> increases, rewards are favorable for the NO and this mechanism is snapshotted to get an average while it attempts to distribute before any snapshot.</p> <p>In normal occasions, it is never possible for a NO to shift his ratio towards receiving more rewards, because it is only expected to decrease the bond. However, if <code>reduced.bond</code> is increased, it will now mean the user can add a new validator while shifting the <code>nodeBond &lt;-&gt; userCapital</code> ratio towards his favor (by providing a higher bond).</p> <p>This also means that more rewards will flow to NO. Usually, whenever this happens, rewards are distributed beforehand in the <code>stake</code> function. A clever NO can abuse such a situation by calling <code>challengeExit</code> (if trusted) or <code>notifyExit</code> for an existing validator, it will then lock reward distribution while the NO is still in old <code>nodeBond &lt;-&gt; userCapital</code>.</p>

	<p>&gt; <code>userCapital</code> ratio, notably, rewards continue to accumulate. Then the NO can queue and stake a new validator (or can even queue before to waste less time). The stake now does not distribute rewards because of the locked validator but it updates the snapshot which means the new ratio is already reflected. If we then wait more time, it will influence the average ratio and result in more rewards [retroactively].</p> <p>c) In cases where <code>reduced.bond</code> is increased, it can happen that it requires <code>bondAmount &gt;= 32 ETH</code> (if a NO has many staked validators which are now underbonded) for new validator addition . This would revert on multiple occasions due to underflows:</p> <pre><code>userQueuedCapital += fullDepositValue - _bondAmount;</code></pre> <p>and it would render it impossible for a NO to actually top up his bond, as topping up would be only possible via adding a new validator while adding the delta required bond amount on top.</p>
<b>Recommendations</b>	Consider never allowing the increase of <code>reduced.bond</code> .
<b>Comments / Resolution</b>	Resolved, Rocketpool team mentioned that A and B are intentional while C has been fixed now due to clamping within the bond requirement calculation.

Issue_35	Non-unique validator addition will result in multiple side-effects, including loss of funds
Severity	High
Description	<p>Within the current concept, it is not enforced that one <code>pubkey/validatorIndex</code> cannot be added multiple times to the same Megapool.</p> <p>This has multiple side-effects:</p> <ul style="list-style-type: none"> <li>a) The second validator addition with the same pubkey will simply increase the effective balance which will then distribute the additional 32 ETH as rewards during the next withdrawal distribution: <ul style="list-style-type: none"> <li>- see <code>is_partially_withdrawable_validator</code> within the consensus specs</li> <li>- This means the 32 ETH will get immediately distributed as rewards via the standard node/user share. Due to the fact that there is an additional node commission <code>[network.node.commission.share]</code>, the NO can execute this attack and immediately receives more ETH than provided.</li> </ul> </li> <li>b) It is possible for a validator to be already during exiting phase, then execute another deposit which is then appended to the withdrawal proof and results in a withdrawal proof of 64 ETH</li> <li>c) The same withdrawal proof can be used multiple times, even if already properly accounted for. This will attempt transfers and increase <code>refundValue</code>. In case with proper timings, these transfers can even be honored in case there is another validator in prestage mode, which can then be in turn used to maliciously increase <code>refundValue</code> to then steal funds as a NO</li> <li>d) Phantom increase of <code>nodeBond</code> and <code>userCapital</code> which is incorrect because these 32 ETH are immediately distributed back (see point a)</li> </ul>

	<p>This issue can furthermore be leveraged to execute more sophisticated attacks, such as stealing funds in case of bond reduction where a zero bond requirement can be repetitively exploited. These highly sophisticated issues use this and other issues as preconditions.</p>
<b>Recommendations</b>	<p>Consider implementing invariant checks for unique pairings between <code>validatorID</code> to <code>publicKey</code> in the megapool. Furthermore, we recommend implementing a global uniqueness-check.</p> <p>We highly recommend a full re-audit for the whole megapool module to ensure high accuracy for post-fixes.</p>
<b>Comments / Resolution</b>	<p>Open. While this issue has been fixed in the following commit:</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/a3bc26ba3cb26a2952e93fc99ddfe20d3f5b4849">https://github.com/rocket-pool/rocketpool/commit/a3bc26ba3cb26a2952e93fc99ddfe20d3f5b4849</a></p> <p>It is currently unclear whether there remain some specific callpaths where this could be bypassed or if there are some introduced issues from this new restriction. The implementation looks straightforward but we are unable to give greenlight on this without a full re-audit of the Megapool.</p>
<b>Resolution 2 / Follow-Up</b>	<p>Resolved, such a uniqueness check is now enforced whenever a new validator is being added.</p> <p>It has to be mentioned that this allows for “burning” a <code>pubKey</code> at [almost] no cost via the <code>dequeue</code> control-flow.</p> <p>A separate medium severity issue has been raised for this.</p>

Issue_36	Usage of partial withdrawal proof for full withdrawal will disrupt accounting
Severity	High
Description	<p>The codebase has the following check within the <code>notifyFinalBalance</code> function:</p> <pre><code>require[_withdrawalSlot &gt;= validator.withdrawableEpoch * slotsPerEpoch, "Not full withdrawal"];</code></pre> <p>This check should ensure that partial withdrawal proofs (reward distributions) cannot be used as full withdrawal proofs.</p> <p>This check is however not 100% secure, as there are multiple scenarios where this can get bypassed but most rely on executing a simple deposit after the first real withdrawal has been executed and use the withdrawal proof of this deposit. It inherently relies on the assumption that a NO calls <code>notifyFinalBalance</code> immediately after the withdrawal has been processed. But this assumption does not reflect reality and furthermore there are states which can further disrupt this assumption.</p> <p>As summarized, this check inherently relies on the fact that the <code>notifyFinalBalance</code> function is called with the first proof but there is no such guarantee for this happening.</p> <p>Under normal circumstances, the risk that this is executed is low, because of the 7d wait time after <code>withdrawable_epoch</code>. That means, even if the attacker would receive a <code>WithdrawalProof</code> right after the owner has received his <code>WithdrawalProof</code> from the regular exit, the owner would need to at least not call <code>notifyFinalBalance</code> for 7 days, which is very unlikely (but can still happen). That alone does however not yet satisfy this for the high severity criteria.</p> <p>However, due to the fact that it is possible to slash a validator that is exiting (which will shift <code>withdrawable_epoch</code> 36d into the future), while <code>notifyExit</code> was already called and <code>withdrawableEpoch</code> was set to pre-slashing <code>withdrawable_epoch</code>, it opens now up the possibility</p>

	<p>for the attacker to call <code>notifyFinalBalance</code> 36d after the <code>withdrawable_epoch</code> in the rocketpool storage for this validator. The clue here is that the original NO can also only call <code>notifyFinalBalance</code> after 36d which means this delay becomes void.</p> <p>Note that this attack always disrupts accounting against the favor of the NO because using a partial withdrawal proof with for example 1 ETH will always result in a large increase of debt for the NO. Hence, an attacker can either execute this attack to damage the NO or simply to profit as RETH holder from the penalty which is then later paid by the NO.</p>
<b>Recommendations</b>	<p>Mitigating this attack will require refactoring the withdrawal process and implement hardening measurements such as validating that the actual <code>withdrawalProof</code> has a reasonable ETH amount.</p> <p>We highly recommend a full re-audit for the whole megapool module to ensure high accuracy for post-fixes.</p>
<b>Comments / Resolution</b>	<p>Open. The Rocketpool team provided the following information:</p> <p>"Mitigated by requiring a longer delay (~30 days) on permissionless final balance notification if the notified balance results in a shortfall of user funds. Now there is a long period of time where the real final balance can be submitted and the fake one cannot. That means there is now a large risk that an attacker's funds simply become a donation as they have to wait a long time to finalise the attack. Given there is low incentive and now a great risk that it does not work, we believe this to be an adequate mitigation of the problem.</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/fadbcfd800cf32a02e0f8e026aab6c5ad453d992">https://github.com/rocket-pool/rocketpool/commit/fadbcfd800cf32a02e0f8e026aab6c5ad453d992</a></p> <p>Also further mitigated by the fact that `notifyFinalBalance` now takes a `ValidatorProof` which confirms the `withdrawable_epoch` at the time of the withdrawal.</p>

	<p><a href="https://github.com/rocket-pool/rocketpool/commit/fadbcfd800cf32a02e0f8e026aab6c5ad453d9">https://github.com/rocket-pool/rocketpool/commit/fadbcfd800cf32a02e0f8e026aab6c5ad453d9</a></p> <p>While this general implementation seems to make sense, we refer to the resolution comment in “Non-unique validator addition will result in multiple side-effects, including loss of funds”</p>
<b>Resolution 2 / Follow-Up</b>	Resolved. Multiple hardening mechanisms have now been implemented which prevent all attacks to our best assessment abilities.

Issue_37	Exited validator can still be added to a Megapool
Severity	High
Description	<p>Deposits to a validator that is already withdrawn increase its balance via <code>apply_pending_deposit</code> and are then swept by <code>get_expected_withdrawals</code> under <code>is_fully_withdrawable_validator[validator, balance, epoch]</code> as soon as possible.</p> <p>Thus, a sequence of 1 ETH [prestake] and then 31 ETH [stake] can both be swept in quick succession (or combined if the prestake has not been swept yet), yielding withdrawal proofs after the (already reached) <code>withdrawable_epoch</code>.</p> <p>When a megapool is created, its creation will not have an impact on the network eth balance until it is assigned. That means, after assignment, the increased <code>userCapital</code> will flow into network eth balance, while it has been decreased on the <code>DepositPool</code>. This mechanism is correct.</p> <p>If now an exited validator is being assigned, this will also follow the same mechanism as above:</p> <ul style="list-style-type: none"> <li>- Decreasing impact of <code>DepositPool</code> user deposits due to withdrawal from <code>RocketVault</code></li> <li>- Enabling impact of assigned validator (<code>userCapital</code> flows into network eth balance)</li> </ul> <p>A problem occurs due to the already described behavior that a deposit post <code>withdrawable_epoch</code> is simply skimmed back. In that scenario, it will be distributed as rewards which flow into various pathways, including the <code>TokenRETH</code> contract via <code>sendToRETH</code>. It is notable that this increase will inherently reflect on the network eth balance.</p> <p>This means, an attacker has reached a state where the megapool's <code>userCapital</code> flows into the network eth balance but at the same time, the skimmed balance also flows partially into it.</p>

	<p>This will inflate the RETH exchange rate with the users' shares while the attacker essentially receives back his full bond.</p> <p>It is noteworthy that the skim back indeed does not happen immediately and there is a risk of being challenged/someone calling <code>notifyExit</code>.</p> <p>However, the attacker can use the trick that the 31 ETH during <code>stake</code> will be used to increase the 1 ETH balance during assignment, combined with the fact that the attacker can implement a delay between assignment and calling <code>stake</code>, such that <code>stake</code> will just be called very shortly before the 1 ETH withdrawal would have been executed but still enough time to process the 31 ETH stake and add it to the 31 ETH. This is slightly tricky but is basically a time-dependent execution which reduces the period between <code>[stake; skim]</code> which then in turn decreases the likelihood of being challenged.</p> <p>It is also important that the 31 eth does not get immediately topped-up, it will need to be processed. Therefore to execute this attack, you need to time exactly the <code>stake</code> call such that the 31 eth still gets processed to the 1 eth sweep in the queue <code>[increase_balance]</code>. All in all, this is very feasible if the deposit queue is short</p>
<b>Recommendations</b>	<p>Consider checking <code>withdrawable_epoch</code> during every aspect of the validator lifecycle addition/stake</p> <p>We highly recommend a full re-audit for the whole megapool module to ensure high accuracy for post-fixes.</p>
<b>Comments / Resolution</b>	<p>Open. The Rocketpool team provided the following comment:</p> <p>Added checks during `stake` that all validator state is correct. This includes checking `withdrawal_credentials`, `withdrawable_epoch`, `exit_epoch`, and `slashed`.</p> <p>Extended `dissolveValidator` to check all above validator fields as well instead of just `withdrawal_credentials`. This means if any of</p>

	<p>those validator fields become “non-compliant” at any time between prestate and stake, the validator can be immediately dissolved. These proofs must be less than approx. 1 hour old now. And because a voluntary exit takes a minimum of 27 hours before `withdrawable_epoch` has passed [‘MIN_VALIDATOR_WITHDRAWABILITY_DELAY’], there is no way to time an exit such that it is possible to pass the ‘stake’ checkpoint and have the validator be skimmed shortly after. There will always be 24+ hours available to challenge the validator as exiting.</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/fadbc800cf832a02e0f8e026aab6c5ad453d992">https://github.com/rocket-pool/rocketpool/commit/fadbc800cf832a02e0f8e026aab6c5ad453d992</a></p> <p>Note: the commit above also includes checking ‘effective_balance’ is equal to 1 ETH which is the usual case of a prestate validator. However, as this value can be increased by a third party with a manual deposit, it could be used to grief NOs by donating 1 ETH into prestate validators allowing them to then be immediately dissolved. So this check was replaced with a check that the ‘effective_balance’ &lt; 32 ETH [‘MIN_ACTIVATION_BALANCE’] and that ‘activation_availability_epoch’ and ‘activation_epoch’ are both ‘FAR_FUTURE_EPOCH’.</p> <p>These checks also prevent an already activated or soon-to-be activated validator from passing the ‘stake’ checkpoint. While I don’t see a specific issue here, there is no need to allow validators in this state to ‘stake’ so we have included this for good measure.</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/b45f9330982137c55d6a72d87294bf675524e2">https://github.com/rocket-pool/rocketpool/commit/b45f9330982137c55d6a72d87294bf675524e2</a></p> <p>We refer to our general resolution comment for this contract.</p>
<b>Resolution 2 / Follow-Up</b>	Resolved, multiple safeguards have been added which now ensure that an exited validator can never consume user provided ETH [via stake]

Issue_38	Credit application due to bond reduction will skew RETH exchange rate
Severity	High
Description	<p>Whenever a new validator is added this will increase <code>nodeBond</code> during <code>newValidator</code> but at this time, this will not yet flow into the total eth balance because the validator is still enqueued and funds are sitting in the <code>DepositPool</code> [total eth balance will only be impacted during assignment].</p> <p>If a bond reduction is happening and <code>reduceBond</code> is called for such a validator, it will immediately call <code>applyCredit</code> which in turn increases the credit balance and thus has a negative impact on the total eth balance.</p>
Recommendations	<p>Consider making the <code>reduceBond</code> call only possible after the prestage period.</p> <p>We highly recommend a full re-audit for the whole megapool module to ensure high accuracy for post-fixes.</p>
Comments / Resolution	<p>Open. The Rocketpool team provided the following comment:</p> <p>We have added a restriction that you cannot `reduceBond` while you have validators in the queue.</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/1ae113625aae5c2a3c0f00af3c2b974f99915a0">https://github.com/rocket-pool/rocketpool/commit/1ae113625aae5c2a3c0f00af3c2b974f99915a0</a></p> <p>We refer to our general resolution comment for this contract.</p>
Resolution 2 / Follow-Up	Resolved, <code>reduceBond</code> cannot be called if there are any unassigned/queued validators.

Issue_39	Manipulation of <code>nodeBond</code> in case of bond reduction allows for stealing majority of accumulated rewards
Severity	High
Description	<p>In <code>newValidator</code>, the contract computes the required total bond if one more validator is added:</p> <pre> <i>RocketNodeDepositInterface</i> rocketNodeDeposit = getRocketNodeDeposit[];  uint256 newBondRequirement = rocketNodeDeposit.getBondRequirement(getActiveValidatorCount( ) + 1);  if [newBondRequirement &gt; nodeBond] {     require[_bondAmount + nodeBond == newBondRequirement, "Bond requirement not met"]; }  require[debt == 0, "Cannot create validator while debt exists"]; } </pre> <p>If the current <code>nodeBond</code> is already <math>\geq</math> the new requirement (e.g., after the DAO previously called <code>reduceBond</code>), the requirement is not enforced. The NO may then choose any <code>_bondAmount</code> for the new validator (including 32 ETH).</p> <p>Rewards splitting in <code>_distributeAmount</code> depends on the current capital ratio at the time of the call:</p> <pre> [, uint256 voterShare, uint256 protocolDAOShare, uint256 rethShare] = rocketNetworkRevenues.calculateSplit[lastDistributionBlock]; unchecked {     uint256 borrowedPortion = _amount * userCapital / (nodeBond + userCapital);     rethRewards = rethShare * borrowedPortion / calcBase;     voterRewards = voterShare * borrowedPortion / calcBase;     protocolDAORewards = protocolDAOShare * borrowedPortion / calcBase; } </pre>

	<pre> nodeRewards = _amount - rethRewards - voterRewards - protocolDAORewards; } </pre> <p>Thus, increasing <code>nodeBond</code> (e.g., by creating a validator with <code>_bondAmount = 32 ETH</code>) immediately increases the NO's share of the already-accrued rewards, in hindsight fashion.</p> <p>The validator created with arbitrary <code>_bondAmount</code> can be removed immediately: <code>dequeue()</code> (still in queue / pre-stake), or <code>dissolveValidator()</code> (pre-stake timeout path).</p> <p>And <code>calculateCapitalDispersal</code> clamps <code>nodeShare</code> by <code>_value</code> (32 ETH), ensuring the operator recovers up to the full 32 ETH just posted.</p> <p>The NO can then convert credit to rETH at 0.01% fee (or reuse the credit), so the bond uplift used to skew the split is returned, while the extra <code>nodeRewards</code> (harvested from the undistributed pot) are kept.</p>
<b>Recommendations</b>	<p>Consider hardening the sanity check of <code>newBondRequirement</code> within the <code>newValidator</code> function and implementing automatic reward distribution when a new validator is added (only after the first existing validator).</p> <p>We highly recommend a full re-audit for the whole megapool module to ensure high accuracy for post-fixes.</p>
<b>Comments / Resolution</b>	<p>Open. The Rocketpool team provided the following comment:</p> <p>We have addressed these 4 findings with the following changes:</p> <ul style="list-style-type: none"> <li>* Fixed issue where arbitrary bond was accepted when NO was overbonded.</li> <li>* Added a 1 ETH minimum bond which is enforced regardless of current bond status.</li> </ul> <p><a href="https://github.com/rocket-pool/rocketpool/commit/9b444078cb54910386c3f93ac08358f4f9">https://github.com/rocket-pool/rocketpool/commit/9b444078cb54910386c3f93ac08358f4f9</a></p>

	<p>c17117</p> <p>Further to the above, we have also improved the way `calculateRewards` works in order to address #33 as well as #39, #41 and #44. It now uses a time-weighted average of the NOs capital ratio since the last distribution instead of the current capital ratio. This time-weighted average uses the “real” capital ratio which does not include queued validator capital so it is only updated at the time of `stake` and not at the time of a deposit. This prevents any situations where a NO can temporarily improve their capital ratio for a reward distribution only to promptly undo it after.</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/588eced354bf60ca6ce1c3b91233a8614479111c">https://github.com/rocket-pool/rocketpool/commit/588eced354bf60ca6ce1c3b91233a8614479111c</a></p> <p>And</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/8f0a026ac371184cfcec3872237ce63bda7bb3c4">https://github.com/rocket-pool/rocketpool/commit/8f0a026ac371184cfcec3872237ce63bda7bb3c4</a></p> <p>Finally, ‘fundsReturned’ is called correctly even when ‘userShare == 0’.</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/9236617334701899893550b11278054d76ab">https://github.com/rocket-pool/rocketpool/commit/9236617334701899893550b11278054d76ab</a></p> <p>We refer to our general resolution comment for this contract.</p>
<b>Resolution 2 / Follow-Up</b>	Resolved, the average <code>capitalRatio</code> is now snapshotted on each ratio change and it is not possible anymore to provide an arbitrary high <code>bondAmount</code> in edge-cases.

Issue_40	Bonded ETH can be manipulated without backing
Severity	High
Description	<p>Rocket's Megapool manages many validators per node-operator and tracks two capital buckets:</p> <ul style="list-style-type: none"> <li>- <code>nodeBond</code> – ETH supplied by the node operator (bonded).</li> <li>- <code>userCapital</code> – ETH borrowed from the protocol (user capital).</li> </ul> <p>When a validator in prestate is dissolved [credentials mismatch, not staked within 28d], dissolveValidator must:</p> <ol style="list-style-type: none"> <li>1. compute how to split the validator deposit between node and user (<code>calculateCapitalDispersal</code>),</li> <li>2. update local accounting (<code>nodeBond</code>, <code>userCapital</code>),</li> <li>3. return appropriate ETH/credits to the <code>RocketDepositPool</code> [via <code>fundsReturned</code> / <code>recycleDissolvedDeposit</code>] and <b>most importantly decrease ETH bonded and borrowed</b>.</li> <li>4. adjust <code>refundValue</code> for node-side funds to be reclaimed late</li> </ol> <p>in <code>dissolveValidator</code>, after computing [<code>nodeShare</code>, <code>userShare</code>] = <code>calculateCapitalDispersal(recycleValue, getActiveValidatorCount() - 1)</code>, the contract may enter the following clause:</p> <pre> if [userShare &gt; 0] {     RocketDepositPoolInterface rocketDepositPool =     getRocketDepositPool();     rocketDepositPool.recycleDissolvedDeposit{value:     userShare}();     rocketDepositPool.fundsReturned[nodeAddress, nodeShare,     userShare]; } </pre> <p>Note how this clause is only triggered if <code>userShare &gt; 0</code>.</p>

If `userShare == 0 but nodeShare > 0` [a realistic outcome when `nodeBond` has been artificially inflated or when `nodeShare` clamps to the recycled value], the megapool never calls `DepositPool.fundsReturned` and as a result of the absence from this call, ETH bonded and ETH borrowed is not decreased.

There is a specific sequence of actions where exactly this state is reached:

- `reduce.bond = 2`
- Alice has 4 validators
- `nodeBond = 16`
- `userShare = 96`
- `ETH bonded = 16`
- `ETH borrowed = 96`

As we can see, the accounting is perfect at this point DAO reduces bond to 2 but Alice does not call `reduceBond` on her Megapool

- a] Alice adds a 5th validator with `bondAmount = 32 ETH`. This is allowed because:
  - i]  $\text{newBondRequirement} = 4 + 4 + 2 + 2 + 2 = 14$

Therefore, we can bypass the bond check and can provide 32 ETH `nodeBond = 48` [**this is the first error in the code that allows for this attack**]

After that, the state is as follows:

- `userShare = 96`
- `ETH bonded = 48`
- `ETH borrowed = 96`

We are now in a state where we have skewed the `nodeBond / userShare` ratio which is usually 4 : 28 and we can call `distribute` to collect accumulated rewards based on our skewed ratio. We have

	<p>already raised this issue itself. <b>This is a precondition to prevent ETH bonded decrease during the following dissolve flow.</b></p> <p>The next step is to:</p> <p>b) Assign the validator via <code>assignFunds</code></p> <p>Now the regular next step would be to call <code>stake</code>, but the exploit path is via a dissolve [which can just be executed because the credentials dont match].</p> <p>The precondition arises here is within this line:</p> <pre><code>[uint256 nodeShare, uint256 userShare] = calculateCapitalDispersal[recycleValue, getActiveValidatorCount() - 1];</code></pre> <ul style="list-style-type: none"> <li>- <code>newBondRequirement = 4 + 4 + 2 + 2 = 12</code></li> <li>- <code>nodeBond = 48</code></li> <li>- <code>nodeShare = 36</code></li> <li>- clamped to <code>nodeShare = 32</code></li> <li>- <code>userShare = 0</code></li> </ul> <p>Now the crux is that the if-clause within <code>dissolveValidator</code>: if <code>[userShare &gt; 0]</code> is only triggered if <code>userShare &gt; 0</code>. <b>This is the second error in the code.</b></p> <p>However, it is zero due to the above reached condition, hence it will not be triggered, hence it will not call <code>DepositPool.fundsReturned</code> and hence it <b>will not decrease ETH bonded</b>.</p> <p>The exploiter now successfully dissolved a megapool and got his bond back while keeping ETH bonded high - allowing to bypass the clamping mechanism in the voting power logic.</p>
<b>Recommendations</b>	<p>Consider introducing multiple hardening measurements such as:</p> <p>a) Ensuring the bond requirement check is not loose</p>

	<p>b] Executing the <code>DepositPool.fundsReturned</code> call even if <code>userShare = 0</code> and <code>nodeShare != 0</code></p> <p>We highly recommend a full re-audit for the whole megapool module to ensure high accuracy for post-fixes.</p>
<b>Comments / Resolution</b>	<p>Open. The Rocketpool team provided the following comment:</p> <p>We have addressed these 4 findings with the following changes:</p> <ul style="list-style-type: none"> <li>* Fixed issue where arbitrary bond was accepted when NO was overbonded.</li> <li>* Added a 1 ETH minimum bond which is enforced regardless of current bond status.</li> </ul> <p><a href="https://github.com/rocket-pool/rocketpool/commit/9b444078cb54910386c3f93ac08358f4fc17117">https://github.com/rocket-pool/rocketpool/commit/9b444078cb54910386c3f93ac08358f4fc17117</a></p> <p>Further to the above, we have also improved the way `calculateRewards` works in order to address #33 as well as #39, #41 and #44. It now uses a time-weighted average of the NOs capital ratio since the last distribution instead of the current capital ratio. This time-weighted average uses the “real” capital ratio which does not include queued validator capital so it is only updated at the time of `stake` and not at the time of a deposit. This prevents any situations where a NO can temporarily improve their capital ratio for a reward distribution only to promptly undo it after.</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/588eced354bf60ca6ce1c3b91233a8614479111c">https://github.com/rocket-pool/rocketpool/commit/588eced354bf60ca6ce1c3b91233a8614479111c</a></p> <p>And</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/588eced354bf60ca6ce1c3b91233a8614479111c">https://github.com/rocket-pool/rocketpool/commit/588eced354bf60ca6ce1c3b91233a8614479111c</a></p>

	<p>pool/rocketpool/commit/8f0a026ac371184cfcec3872237ce63bda7bb3c4</p> <p>Finally, `fundsReturned` is called correctly even when `userShare == 0`.</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/9236617334701899893550b11278054d76ab">https://github.com/rocket-pool/rocketpool/commit/9236617334701899893550b11278054d76ab</a></p> <p>We refer to our general resolution comment for this contract.</p>
<b>Resolution 2 / Follow-Up</b>	Resolved, <code>DepositPool.fundsReturned</code> is now called without any exemption.

Issue_41	1 ETH drain due to zero bond requirement
Severity	High
Description	<p>Currently, it is possible to steal 1 ETH from a very specific megapool setup.</p> <p>There is an edge case during bond reduction where a validator can be added with zero bond requirement in case he has already 3 validators and the bond is reduced to 2:</p> <ul style="list-style-type: none"> <li>- nodeBond = 12</li> <li>- userCapital = 84</li> <li>- reduce.bond = 2</li> <li>- 5th validator added <ul style="list-style-type: none"> <li>- nodeBond = 12</li> <li>- userCapital = 116</li> </ul> </li> <li>- nodeBondRequired = <math>4 + 4 + 2 + 2 + 2</math></li> <li>- In that case, newValidator passes with bondAmount = 0, the user essentially did not provide a bond. The validator is now in the queue</li> <li>- The validator is now assigned and the 32 ETH is deposited to MegapoolDelegator. 1 ETH is prestaked to a pubkey which points to attacker credentials</li> <li>- Attacker dissolves the validator <ul style="list-style-type: none"> <li>- nodeShare = 0</li> <li>- userShare = 32</li> <li>- nodeBond = 12</li> <li>- userCapital = 86</li> </ul> </li> </ul> <p>After that, the state is simply reversed, while the attacker stole 1 ETH via the prestake call to the validator which points to the attacker's own credentials.</p>
Recommendations	<p>Consider implementing a more strict check within the <code>bondRequirement</code> logic. It should not be possible to create a validator without any bond.</p> <p>We highly recommend a full re-audit for the whole megapool</p>

	<p>module to ensure high accuracy for post-fixes.</p>
Comments / Resolution	<p>Open. The Rocketpool team provided the following comment:</p> <p>We have addressed these 4 findings with the following changes:</p> <ul style="list-style-type: none"> <li>* Fixed issue where arbitrary bond was accepted when NO was overbonded.</li> <li>* Added a 1 ETH minimum bond which is enforced regardless of current bond status.</li> </ul> <p><a href="https://github.com/rocket-pool/rocketpool/commit/9b444078cb54910386c3f93ac08358f4fc17117">https://github.com/rocket-pool/rocketpool/commit/9b444078cb54910386c3f93ac08358f4fc17117</a></p> <p>Further to the above, we have also improved the way `calculateRewards` works in order to address #33 as well as #39, #41 and #44. It now uses a time-weighted average of the NOs capital ratio since the last distribution instead of the current capital ratio. This time-weighted average uses the "real" capital ratio which does not include queued validator capital so it is only updated at the time of `stake` and not at the time of a deposit. This prevents any situations where a NO can temporarily improve their capital ratio for a reward distribution only to promptly undo it after.</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/588eced354bf60ca6ce1c3b91233a8614479111c">https://github.com/rocket-pool/rocketpool/commit/588eced354bf60ca6ce1c3b91233a8614479111c</a></p> <p>And</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/8f0a026ac371184cfcec3872237ce63bda7bb3c4">https://github.com/rocket-pool/rocketpool/commit/8f0a026ac371184cfcec3872237ce63bda7bb3c4</a></p> <p>Finally, `fundsReturned` is called correctly even when `userShare ==</p>

	<p>0`.</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/9236617334701899893550b11278054d76ab">https://github.com/rocket-pool/rocketpool/commit/9236617334701899893550b11278054d76ab</a></p> <p>We refer to our general resolution comment for this contract.</p>
<b>Resolution 2 / Follow-Up</b>	Resolved, it is now ensured that a minimum bond of <code>prestakeAmount</code> [1 ETH] is provided upon validator addition

Issue_42	<p>Permanently locked <code>userShare</code> in case of validator creation with zero bond due to underflow revert within <code>dissolveValidator</code></p>
Severity	High
Description	<p>In the issue above, we explained how it is possible for an attacker to steal the 1 ETH pre-stake amount in a very specific edge-case. The above condition can be further exploited, as essentially now 31 ETH remain in the Megapool while the <code>stake</code> function cannot be invoked because credentials do not match.</p> <p>Naturally, the next step would be to dissolve the validator. This step however reverts due to an underflow:</p> <pre>refundValue += nodeShare - preStakeValue;</pre> <p>While:</p> <pre>nodeShare = 0 preStakeValue = 1</pre> <p>Therefore, <code>dissolve</code> can never be called which means that the <code>userShare/userCapital</code> will remain permanently stuck within the Megapool contract and the exploiter locked 31 ETH permanently [they remain as <code>assignedValue</code>]. There may be scenarios how the megapool owner can trigger future validator additions which reverse that state. However, that still means the megapool owner can deliberately prevent this.</p>
Recommendations	<p>Consider ensuring that it is never possible to add a validator while the provided bond amount is below the <code>preStakeValue</code>. Furthermore, we recommend handling potential underflows within the said function. Fuzzing is mandatory to identify potential other states where such a revert can be enforced.</p> <p>We highly recommend a full re-audit for the whole megapool module to ensure high accuracy for post-fixes.</p>
Comments / Resolution	<p>Open. The Rocketpool team provided the following comment:</p> <p>We have addressed these 4 findings with the following changes:</p>

\* Fixed issue where arbitrary bond was accepted when NO was overbonded.

\* Added a 1 ETH minimum bond which is enforced regardless of current bond status.

<https://github.com/rocket-pool/rocketpool/commit/9b444078cb54910386c3f93ac08358f4f9c17117>

Further to the above, we have also improved the way `calculateRewards` works in order to address #33 as well as #39, #41 and #44. It now uses a time-weighted average of the NOs capital ratio since the last distribution instead of the current capital ratio. This time-weighted average uses the “real” capital ratio which does not include queued validator capital so it is only updated at the time of ‘stake’ and not at the time of a deposit. This prevents any situations where a NO can temporarily improve their capital ratio for a reward distribution only to promptly undo it after.

<https://github.com/rocket-pool/rocketpool/commit/588eced354bf60ca6ce1c3b91233a8614479111c>

And

<https://github.com/rocket-pool/rocketpool/commit/8f0a026ac371184cfcec3872237ce63bda7bb3c4>

Finally, ‘fundsReturned’ is called correctly even when ‘userShare == 0’.

<https://github.com/rocket-pool/rocketpool/commit/9236617334701899893550b11278054d76ab>

	We refer to our general resolution comment for this contract.
<b>Resolution 2 / Follow-Up</b>	Resolved, the logic has been refactored such that an underflow in this scenario becomes mathematically impossible.

Issue_43	Zero <code>validatorIndex</code> proof can be used to steal funds from the protocol
Severity	High
Description	<p>Within the lifecycle of a validator creation, the <code>validatorIndex</code> of a validator is only assigned during the <code>stake</code> function:</p> <pre><code>validator.validatorIndex = _validatorIndex;</code></pre> <p>If a validator has not staked, <code>validatorIndex</code> remains zero.</p> <p>Furthermore, there is a path for dissolved validators within the <code>notifyFinalBalance</code> function, while a dissolved validator is considered as having credentials that are not matching, it appears that this path is just wrong.</p> <p>These two flaws can be exploited by the fact that it is possible to call <code>notifyFinalBalance</code> with the withdrawal proof for validator with index 0 (once this validator has indeed exited - at this point the validator is still staking).</p> <p>The exploit path is simple:</p> <ul style="list-style-type: none"> <li>- Malicious user adds a validator with wrong credentials and never stakes it</li> <li>- The validator will get enqueued and the 1 ETH pre-stake is happening</li> <li>- The validator will get dissolved because credentials do not happen</li> <li>- <code>notifyExit</code> is called with the validator proof for the zero index validator (if he has exited). The check matches because by default, <code>validatorIndex</code> for a non-staked validator is zero. There is furthermore no <code>pubKey</code> check.</li> <li>- <code>notifyFinalBalance</code> is called with the withdrawal proof for the exited validator with <code>validatorIndex</code> zero. The check matches because by default, <code>validatorIndex</code> for a non-staked validator is zero. There is furthermore no <code>pubKey</code> check.</li> <li>- <code>refundValue</code> is increased while the contract has not received</li> </ul>

	<p>any balance</p> <p>The increased <code>refundValue</code> can then be consumed by simply creating another validator with wrong credentials, waiting until it gets assigned and then exploit the idle 31 balance post-assignment.</p> <p>Using this approach, it is possible to swiftly drain all user funds that are sitting in the DepositPool.</p>
<b>Recommendations</b>	<p>We recommend the following hardening applications:</p> <ul style="list-style-type: none"> <li>a) Remove dissolve logic</li> <li>b) Ensure that pubkey matches during <code>notifyExit</code> and <code>notifyFinalBalance</code></li> <li>c) Add withdrawal credential match during <code>notifyExit</code> and <code>notifyFinalBalance</code></li> </ul> <p>We highly recommend a full re-audit for the whole megapool module to ensure high accuracy for post-fixes.</p>
<b>Comments / Resolution</b>	<p>Open. The Rocketpool team provided the following comment:</p> <p>We have entirely removed the use of `validatorIndex` and instead rely on pubkeys to uniquely identify validators. This is possible now that `notifyFinalBalance` takes a `ValidatorProof` as well as a `WithdrawalProof`.</p> <p>Before we were relying on the proven `validatorIndex` from the `stake` operation and storing that value to check in the `notifyFinalBalance` function. But by requiring a `ValidatorProof` on final balance, we no longer need to rely on the stored value. This solves the issue of `validatorIndex` zero referring to a real validator and also solves the issue that dissolved validators did not have a `validatorIndex` set as `stake` was never called.</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/fadbc800cf832a02e0f8e026aab6c5ad453d99">https://github.com/rocket-pool/rocketpool/commit/fadbc800cf832a02e0f8e026aab6c5ad453d99</a></p>

	We refer to our general resolution comment for this contract.
<b>Resolution 2 / Follow-Up</b>	Resolved, a <a href="#">pubkey</a> check has been implemented within all state transition functions in the MegapoolManager contract.

<b>Issue_44</b>	Beaconchain slashing withdrawal delays allows debt accrual and repayment bypass [FOLLOWUP]
<b>Severity</b>	Medium
<b>Description</b>	<p>The <code>notifyFinalBalance</code> function releases bond capital per-validator and triggers immediate <code>_claim()</code> for node operators, bypassing the <code>numExitingValidators</code> check that only protects <code>distribute()</code>.</p> <p>When one validator is slashed (36-day withdrawal delay) and others exit normally (~5 days), NO can call <code>notifyFinalBalance</code> for non-slashed validators and extract their bond shares via the auto-claim.</p> <p>By the time the slashed validator's shortfall materializes as debt (which would prevent withdrawal crediting), withdrawals have already been processed and <code>nodeBond</code> is insufficient to cover losses.</p> <p>Example:</p> <ol style="list-style-type: none"> <li>1. 4 validators with 4 ETH bond each.</li> <li>2. V1 slashed 50%.</li> <li>3. NO extracts 12 ETH from V2/V3/V4 exits.</li> <li>4. V1 returns 16 ETH instead of 32 ETH, creating 12 ETH debt with only 4 ETH bond remaining</li> <li>5. rETH holders bear unrecoverable loss when NO had sufficient capital coverage.</li> </ol> <p>Please note that this issue is tied to the fact that the last validator can cause unpaid debt (which was already raised and known). This is just an extended control-flow which slightly increases the impact</p>
<b>Recommendations</b>	<p>Consider acknowledging this issue as a fix for that would be non-trivial and would essentially mean the debt application during <code>_notifyFinalBalance</code> is refactored.</p> <p>It can be considered as a design constraint that the last validator can never be forced to repay debt.</p>
<b>Comments / Resolution</b>	Acknowledged.

Issue_45	Staleness of capitalRatio in case of <code>nodeBond = 0</code> while validators are staked [FOLLOWUP]
Severity	Medium
Description	<p>The Megapool splits rewards using a capital-ratio model: <code>capitalRatio = [userCapital + nodeBond] / nodeBond</code>. This ratio is snapshotted in <code>RocketNetworkRevenues</code> via <code>_snapshotCapitalRatio()</code>, and reward distribution uses a time-weighted average of these snapshots since <code>lastDistributionTime</code>. The implementation assumes that if <code>nodeBond == 0</code>, the node has exited all validators, and therefore <code>_snapshotCapitalRatio()</code> skips updating capital ratio in that case.</p> <p><code>_calculateCapitalDispersal()</code> computes <code>nodeShare</code> based on <code>effectiveBond = nodeBond + nodeQueuedBond</code>, but clamps <code>nodeShare</code> to <code>nodeBond</code> before it is applied (<code>nodeBond -= nodeShare</code>). When <code>nodeQueuedBond</code> alone is sufficient to satisfy the bond requirement for the remaining active validator count, dispersal during exit finalization can legitimately set <code>nodeShare == nodeBond</code>, reducing <code>nodeBond</code> to <code>zero</code> even though other validators remain staked (and/or additional validators remain queued and counted as active).</p> <p>This violates the implicit invariant that <code>nodeBond == 0</code> implies “no active validators,” and it causes <code>_snapshotCapitalRatio()</code> to stop updating the node’s capital ratio while the Megapool continues accruing rewards.</p> <p>After reaching a state where staked validators exist but <code>nodeBond == 0</code>, capital ratio snapshots become stale and reward distributions use an outdated finite ratio. This can overpay the node (lower <code>computedBorrowedPortion</code>) relative to the intended model for a zero-bond state, while the node’s capital is effectively shifted into queued bond (not exposed to validator slashing). This results in a risk/reward mismatch and potentially material reward misallocation over time, especially if the stale ratio persists until queued validators are eventually assigned/staked/dequeued.</p>

	<p>Example:</p> <ul style="list-style-type: none"> <li>- 3 staked validators: <code>nodeBond=12, userCapital=84</code></li> <li>- queue 3 more: <code>nodeQueuedBond=12, userQueuedCapital=84</code>            [these validators are counted as “active” because  <code>getActiveValidatorCount = numValidators - numInactiveValidators</code>]</li> <li>- governance reduces post-first bond to 1</li> <li>- finalize exit of one validator (<code>_notifyFinalBalance</code>):           <ul style="list-style-type: none"> <li>- <code>_newValidatorCount = active-1 = 5</code></li> <li>- <code>newBondRequirement[5] = 4 + 1 + 1 + 1 + 1 = 8</code></li> <li>- <code>effectiveBond = 12 + 12 = 24</code></li> <li>- <code>nodeShare = 24 - 8 = 16 \rightarrow clamped to nodeBond=12</code></li> <li>- <code>userShare = 32 - 12 = 20</code></li> <li>- after accounting:               <ul style="list-style-type: none"> <li>- <code>nodeBond = 0</code></li> <li>- <code>userCapital = 64</code></li> <li>- <code>nodeQueuedBond = 12</code> remains</li> </ul> </li> </ul> </li> </ul> <p><b>Result:</b> staked validators remain, but <code>nodeBond == 0</code>.</p> <p>Even if the node still has 12 ETH “nominally” locked as queued bond, queued bond is not equivalently at risk as staked bond (it sits in the DepositPool/Vault domain rather than being exposed to validator slashing). So there is still a meaningful economic gain: reduced risk while retaining reward economics tied to a stale ratio.</p>
<b>Recommendations</b>	<p>A simple fix for this issue without refactoring code is quite tricky, especially given the fact that during this last round, code changes should be limited to an absolute minimum. While an option might be to change the fundamental mechanism of <code>nodeQueuedBond</code> impact on <code>effectiveBond</code>, such a change would impact lifecycles in a non-trivial way and would most likely result in other bugs.</p> <p>Therefore, we recommend acknowledging this issue for the time</p>

	being and being sensitive with any potential bond reductions, as the root-cause for this issue relies on bond reduction while having queued validators.
<b>Comments / Resolution</b>	Resolved.

Issue_46	Severe average capitalRatio distortion in edge-case [FOLLOWUP]
Severity	Medium
Description	<p>Capital ratio is computed as <code>[userCapital + nodeBond] / nodeBond</code>. If <code>nodeBond</code> becomes very small, <code>capitalRatio</code> becomes very large. <code>RocketNetworkRevenues</code> time-averages the stored <code>capitalRatio</code>, and <code>RocketMegapoolDelegate.calculateRewards</code> uses <code>1 / averageCapitalRatio</code> to compute the bonded portion of rewards.</p> <p><code>_calculateCapitalDispersal</code> can reduce <code>nodeBond</code> down to the residual <code>newBondRequirement - nodeQueuedBond</code> (or to zero if the residual is non-positive), because it uses <code>effectiveBond = nodeBond + nodeQueuedBond</code> to determine how much bond can be released from <code>nodeBond</code> while still meeting the post-event requirement.</p> <p>If edge-cases allow <code>newBondRequirement</code> to be arbitrarily close to <code>nodeQueuedBond</code>, the remaining <code>nodeBond</code> can be arbitrarily small but non-zero, creating extremely large capital ratios. When such a large ratio exists temporarily (especially across a long interval where distribution is blocked by exiting/locked validators), the time-weighted average capital ratio can swing heavily, and due to the non-linear <math>1/x</math> mapping used in <code>calculateRewards</code>, payout outcomes become highly sensitive to the timing of ratio updates and distribution.</p> <p>Example:</p> <p><code>[4,4,4] = 3 staked validators [capitalRatio = 8]</code></p> <ul style="list-style-type: none"> <li>- <code>nodeBond = 12</code></li> <li>- <code>userCapital = 84</code></li> </ul> <p>queue 3 more validators</p> <ul style="list-style-type: none"> <li>- <code>nodeBond = 12</code></li> <li>- <code>userCapital = 84</code></li> <li>- <code>nodeBondQueued = 12</code></li> <li>- <code>userCapitalQueued = 84</code></li> </ul> <p>DAO sets reduced.bond to 2.0000000025</p>

	<p>Exit validator 1</p> <ul style="list-style-type: none"> <li>- bondRequired = [4 , 2.0000000025, 2.0000000025, 2.0000000025, 2.0000000025] = 12.00000001</li> <li>- effectiveBond = 24</li> <li>- nodeShare = 11.99999999</li> <li>- userShare = 20.00000001</li> <li>- nodeBond = 0.00000001</li> <li>- userCapital = 63.99999999</li> <li>- nodeBondQueued = 12</li> <li>- userCapitalQueued = 84</li> </ul> <p>It now has a super small capital ratio while excluding any queued funds</p> <ul style="list-style-type: none"> <li>- &gt; capitalRatio = (userCapital + nodeBond) * calcBase / nodeBond</li> <li>- &gt; [63.99999999e18 + 0.00000001e18] * 1e18 / 0.00000001e18</li> <li>- &gt; <b>capitalRatio = 6.4e27</b></li> </ul>
<b>Recommendations</b>	Consider keeping the reduced.bond value % 1e18.
<b>Comments / Resolution</b>	Resolved.

Issue_47	DoS of reward claim due to edge-case with two <code>notifyExit</code> calls
Severity	Medium
Description	<p>Rewards cannot be claimed as soon as <code>withdrawableEpoch</code> is reached. This mechanism is correct but does not treat a specific edge-case with two withdrawals happening.</p> <p>In the scenario where <code>soonestWithdrawableEpoch</code> is set to the first <code>withdrawable_epoch</code> while there is at the same time a later withdrawal for another validator, rewards will not be claimable as soon as <code>withdrawableEpoch</code> is reached. However, they should be again claimable between <code>[soonestWithdrawableEpoch; *nextWithdrawableEpoch]</code> after the first exit is finalized. However, this is not possible because of a lack of <code>soonestWithdrawableEpoch</code> reset if there is another pending withdrawal:</p> <pre><code>if [numExitingValidators == 0] {     soonestWithdrawableEpoch = 0; }</code></pre> <p>This is because the first exit does not reset <code>soonestWithdrawableEpoch</code> and now it will still point to the epoch which is tied to a validator that has already exited, preventing legitimate reward claims until the next <code>withdrawal_epoch</code> is reached. As long as the next <code>withdrawableEpoch</code> has not been reached, reward claims should still be permitted.</p> <p>The impacts here may prevent large account holders from processing any rewards across the entire megapool. Large accounts may total huge amounts of inaccessible funds daily.</p> <p>After further internal discussion, this issue has been raised from medium to high severity because of the above described impact.</p>
Recommendations	This requires a refactoring of the approach to ensure <code>soonestWithdrawableEpoch</code> is correctly reset once it has been reached and it should not rely on <code>numExitingValidators</code> as a source of truth because this exact scenario may happen.

	We highly recommend a full re-audit for the whole megapool module to ensure high accuracy for post-fixes.
<b>Comments / Resolution</b>	Acknowledged. This is a design choice. After further assessment, the issue severity has been downgraded from high severity to medium severity.

<b>Issue_48</b>	Ambiguous behavior due to slashing after exit can result in malicious action and deliberate DoS of rewards
<b>Severity</b>	Medium
<b>Description</b>	<p>When a validator is being slashed after he has exited, the <code>withdrawable_epoch</code> on the BC for the validator is shifted back. At this point, there are actually two possible validator states:</p> <ul style="list-style-type: none"> <li>a) The current one with the correct <code>withdrawable_epoch</code></li> <li>b) The past one with the pre-slashed <code>withdrawable_epoch</code></li> </ul> <p>It may be possible that <code>notifyExit</code> has not yet been called and in this scenario it is expected that the NO calls this function with the correct validator state.</p> <p>However, it is possible that a malicious actor observes this situation and calls the function with the pre-slashed state. The impact of this will be a partial DoS of reward distribution [as mentioned in another issue].</p>
<b>Recommendations</b>	<p>There is no trivial fix for this issue with the current design.</p> <p>We highly recommend a full re-audit for the whole megapool module to ensure high accuracy for post-fixes.</p>
<b>Comments / Resolution</b>	<p>Open. The Rocketpool team provided the following comment:</p> <p>Added validator proof to `notifyFinalBalance` to also check that `withdrawal_epoch` is actually in the past at the time of the withdrawal.</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/fadbcfd800cf32a02e0f8e026aab6c5ad453d9">https://github.com/rocket-pool/rocketpool/commit/fadbcfd800cf32a02e0f8e026aab6c5ad453d9</a></p> <p>We refer to our general resolution comment for this contract.</p>
<b>Resolution 2 / Follow-Up</b>	Acknowledged, the logic has been refactored and reward distribution now solely relies on <code>numExitingValidators</code> . If a validator has been slashed, it is still possible [and even expected] to call the

`notifyExit` function which then increases `numExitingValidators`, while the `withdrawable_epoch` for the validator is  $> 36d$  in the future.

This can be considered as a hardening design decision but impacts need to be kept in mind.

<b>Issue_49</b>	Reward payout lag due to slashing will cause side-effects in follow-up distributions [RETH, voterAmount, claimDAO]
<b>Severity</b>	Medium
<b>Description</b>	Whenever a validator is being slashed, the <code>withdrawable_epoch</code> will be shifted by 36 days. This will expose an inherent reward distribution lag if an already notified validator is being slashed, <code>assoonestWithdrawableEpoch</code> will point to the pre-slashing <code>withdrawableEpoch</code> and rewards are force-delayed which will have further side-effects on reward distribution flows (accumulating of rewards without payout)
<b>Recommendations</b>	We highly recommend a full re-audit for the whole megapool module to ensure high accuracy for post-fixes.
<b>Comments / Resolution</b>	<p>Open. The Rocketpool team provided the following comment:</p> <p>This is somewhat mitigated by the use of a time-weighted average of capital ratio for reward distribution. The delay in rewards is not fixable and an inherent part of the Megapool design.</p> <p>But by using a time-weighted average of the NOs capital ratio, the timing of reward distribution no longer materially affects the rewards outcome.</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/588eced354bf60ca6ce1c3b91233a8614479111c">https://github.com/rocket-pool/rocketpool/commit/588eced354bf60ca6ce1c3b91233a8614479111c</a></p> <p>And</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/8f0a026ac371184cfcec3872237ce63bda7bb3c4">https://github.com/rocket-pool/rocketpool/commit/8f0a026ac371184cfcec3872237ce63bda7bb3c4</a></p> <p>We refer to our general resolution comment for this contract.</p>
<b>Resolution 2 / Follow-Up</b>	Partially resolved, while now an average <code>capitalRatio</code> is implemented which partially offsets some impacts of reward delay, the fundamental fact still remains.

Issue_50	Side effects due to lack of reward claim during <code>notifyFinalBalance</code>
Severity	Medium
Description	<p>The <code>notifyFinalBalance</code> function does not automatically claim rewards. Rewards are based on the bond to user share ratio.</p> <p>This is an issue because in certain scenarios, the <code>notifyFinalBalance</code> function will have an inherent impact on the node / user ratio in the case where <code>reduce.bond</code> has been changed.</p> <p>Usually there is a 12.5% ratio [4/28] but in case of <code>reduce.bond</code> change, there are edge-cases such as:</p> <ul style="list-style-type: none"> <li>- validator 1= 4 / 28 [12.5% to NO]</li> <li>- validator 2 = 8 / 56 [12.5% to NO]</li> <li>- reduced bond to 3 (governance)</li> <li>- validator 3 = 11 / 85 [ 11.4% to NO]</li> <li>- Now we accumulate rewards with 11.4% share</li> <li>- We never claim rewards but exit validator 3, this will simply reset the ratio to 12.5%, without claiming</li> </ul> <p>As one can see, rewards should be distributed with 11.4% share but that didn't happen. Now the share is back to 12.5% and NO receives more share when claiming. Likewise, the addition of a third validator in that scenario will not claim any rewards and now the ratio will be 11.4% while it was previously 12.5%.</p> <p>This issue can even be escalated in case a validator exits, calls <code>notifyExit</code> to set <code>withdrawableEpoch</code>, then gets slashed which increases <code>withdrawable_epoch</code> on the beacon chain without setting <code>withdrawableEpoch</code> within the RocketPool storage. The effect is that this force-delays rewards with a 11.4% share, while they can be only claimed much later with 12.5% share.</p>
Recommendations	Consider invoking automatic reward claims on several state transitions such as adding a validator and removing a validator. It has to be noted that this should not happen during the addition of the very first validator.

	<p>We highly recommend a full re-audit for the whole megapool module to ensure high accuracy for post-fixes.</p>
<b>Comments / Resolution</b>	<p>Open. The Rocketpool team provided the following comment:</p> <p>We cannot always invoke automatic reward claims on these state transitions as the Megapool contract balance might contain multiple final balance withdrawals.</p> <p>For example, if two validators exited at the same time and both 32 ETH balances return to the Megapool.</p> <p>Forcing distribution on final balance of one would distribute the other 32 ETH as rewards.</p> <p>We have mitigated this problem by using a time-weighted average of the Megapool's capital ratio since the last distribution instead. This is necessary because otherwise an attacker with a lot of validators could purposefully exit validators slowly over time resulting in blocking of distribution for an extended period of time.</p> <p>They could then eventually distribute all the rewards during that period at the final capital ratio. As bond will always decrease with more validators, the final ratio will always be in the NO's favour.</p> <p>So we need to instead keep track of the average capital ratio over time and distribute all rewards at the average ratio since the previous distribution. Similarly to how we handle the time-weighted average of the fees.</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/588eced354bf60ca6ce1c3b91233a8614479111c">https://github.com/rocket-pool/rocketpool/commit/588eced354bf60ca6ce1c3b91233a8614479111c</a></p>

	<p>We have also included an automatic reward distribution in the situations where it is possible to do so (i.e. during a state transition if there are no locked or exiting validators we can distribute before the ratio changes).</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/8f0a026ac371184cfcec3872237ce63bda7bb3">https://github.com/rocket-pool/rocketpool/commit/8f0a026ac371184cfcec3872237ce63bda7bb3</a></p> <p>We refer to our general resolution comment for this contract.</p>
<b>Resolution 2 / Follow-Up</b>	Resolved, an average <code>capitalRatio</code> mechanism has been implemented. However, there are still certain scenarios where reward payouts can be slightly gamed.

<b>Issue_51</b>	Call of <code>distribute</code> before any active validator will skew <code>lastDistributionBlock</code> and related share ratio calculation
<b>Severity</b>	Medium
<b>Description</b>	Currently, it is possible to call <code>distribute</code> without any active validator which then sets <code>lastDistributionBlock</code> . The result of this setting is that all reward related shares begin their average calculation from that time on, while it is desired that this only happens after the first validator has staked.
<b>Recommendations</b>	Consider preventing <code>distribute</code> to be called when <code>lastDistributionBlock</code> is unset.
<b>Comments / Resolution</b>	<p>Open. The Rocketpool team provided the following comment:</p> <p>Fixed by preventing 'distribute' when 'lastDistributeBlock' is zero.</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/aa4c4471c3df80d64514d12597831e2a869ed2c8">https://github.com/rocket-pool/rocketpool/commit/aa4c4471c3df80d64514d12597831e2a869ed2c8</a></p> <p>We refer to our general resolution comment for this contract.</p>
<b>Resolution 2 / Follow-Up</b>	Resolved.
<b>Resolution 3</b>	<p>We have identified an edge-case where this still is true:</p> <ul style="list-style-type: none"> <li>a) Assign validator A</li> <li>b) Dissolve a validator A, set <code>lastDistributionTime &gt; 0</code></li> </ul> <p>Now there is no staked validator but <code>distribute</code> can be called, resulting in the same issue. The issue status will be changed from resolved to partially resolved because this execution requires some step in between to succeed.</p>

<b>Issue_52</b>	Lack of dissolve penalty allows to bloat the queue
<b>Severity</b>	Medium
<b>Description</b>	<p>Currently, there is no penalty for dissolving a megapool, as any dissolved megapool will grant 100% of the provided funds back to the owner.</p> <p>This opens up an attack-vector where a malicious user can create a large amount of validators (assuming he has the capital) and essentially bloat the queue, as these validators will be dissolved.</p> <p>The attacker bloated the queue and delayed legitimate validator creations while only temporarily locking the capital.</p>
<b>Recommendations</b>	<p>Consider implementing a penalty for dissolving a validator. This control flow must be fully audited.</p> <p>We highly recommend a full re-audit for the whole megapool module to ensure high accuracy for post-fixes.</p>
<b>Comments / Resolution</b>	<p>Open. The Rocketpool team provided the following comment:</p> <p>Implemented a dissolve penalty as recommended.</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/c8e539660e02a6a7baa11986de7bbe2a62fe06">https://github.com/rocket-pool/rocketpool/commit/c8e539660e02a6a7baa11986de7bbe2a62fe06</a></p> <p>We refer to our general resolution comment for this contract.</p>
<b>Resolution 2 / Follow-Up</b>	Resolved.

<b>Issue_53</b>	Final rewards will be only allocated to NO
<b>Severity</b>	Medium
<b>Description</b>	<p>In the scenario where the last validator is exiting, it will reset <code>nodeBond</code> and <code>userCapital</code> to zero.</p> <p>There are multiple scenarios with unclaimed rewards after exiting, which can happen because a validator still receives rewards until <code>exit_epoch</code> or if there are simply undistributed rewards.</p> <p>In that scenario, these rewards can be claimed after the final exit. However, instead of distributing them fairly, they will just be distributed to the NO:</p> <pre><code>if [totalCapital == 0] return [amount, 0, 0, 0];</code></pre>
<b>Recommendations</b>	<p>Consider automatically invoking the <code>distribute</code> function before all state transitions that alter the distribution ratio.</p> <p>We highly recommend a full re-audit for the whole megapool module to ensure high accuracy for post-fixes.</p>
<b>Comments / Resolution</b>	<p>Open. The Rocketpool team provided the following comment:</p> <p>We have mitigated this problem by using a time-weighted average of the Megapool's capital ratio since the last distribution instead.</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/588eced354bf60ca6ce1c3b91233a8614479111c">https://github.com/rocket-pool/rocketpool/commit/588eced354bf60ca6ce1c3b91233a8614479111c</a></p> <p>And</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/8f0a026ac371184cfcec3872237ce63bda7bb3c">https://github.com/rocket-pool/rocketpool/commit/8f0a026ac371184cfcec3872237ce63bda7bb3c</a></p> <p>We refer to our general resolution comment for this contract.</p>

<b>Resolution 2 / Follow-Up</b>	<p>Resolved, the average <code>capitalRatio</code> does only help in this scenario where there are the rewards after the exit, as these will be distributed via <code>distribute</code>, using the <code>capitalRatio</code> mechanism.</p> <p>Rewards which are distributed alongside the exit are commingled into <code>_amountInGwei</code>.</p> <p>These specific rewards are now allocated to the user side:</p> <pre><code>uint256 toUser = withdrawalBalance - toNode;</code></pre> <p>A separate issue will be raised for this concern.</p>
---------------------------------	---

Issue_54	Loss of prestakeValue in case of missing stake call for 28 days
Severity	Medium
Description	The <code>dissolveValidator</code> function is callable 28 days after a validator has been assigned. It essentially allows to dissolve a validator that has not yet staked for whatever reason. If the validator has indeed correct credentials (which he should have, otherwise it would have been dissolved way earlier than after 28 days), the 1 ETH prestaked amount will simply be lost.
Recommendations	<p>Consider keeping this scenario in mind and implementing a warning on the frontend or optionally scan all validators and invoke <code>stake</code> manually in case the validator itself is not doing it within a reasonable timeframe.</p> <p>We highly recommend a full re-audit for the whole megapool module to ensure high accuracy for post-fixes.</p>
Comments / Resolution	<p>Acknowledged. The Rocketpool team considers this as a non-issue and provided the following comment:</p> <p>If the validator has correct credentials but was dissolved, they can recover the 1 ETH by depositing an additional 31 ETH into the validator manually, then exiting it. The funds will be returned to the Megapool and can then be distributed 100% to the node operator.</p> <p><i>While we agree that this is theoretically possible, we need to keep in mind that not every user has 31 ETH that he can deposit for a certain period. This would also not be possible to retrieve with a flashloan because it spans over multiple days, including being exposed to the price risk of ETH. Therefore, we consider this as a valid issue.</i></p>

<b>Issue_55</b>	Rewards can be locked due to ambiguous validator state in slashing case
<b>Severity</b>	Medium
<b>Description</b>	<p>It is possible that a validator is slashed after he has already exited but not yet called <code>notifyExit</code>.</p> <p>This means an attacker can use the validator state pre-slashing to call <code>notifyExit</code> which then in turn uses the pre-slashing <code>withdrawable_epoch</code> which then in turn results in an earlier locked distribute function than expected (it should only be locked once the actual <code>withdrawable_epoch</code> is reached, which is represented by the post-slashing <code>withdrawable_epoch</code>)</p>
<b>Recommendations</b>	<p>A fix for this is non-trivial as it requires a permission change of <code>notifyExit</code> which comes with other side-effects.</p> <p>We highly recommend a full re-audit for the whole megapool module to ensure high accuracy for post-fixes.</p>
<b>Comments / Resolution</b>	<p>Open. The Rocketpool team provided the following comment:</p> <p>We have fixed this by having `notifyFinalBalance` take a `ValidatorProof` as well as a `WithdrawalProof`. This guarantees that the withdrawal used in the final balance notification took place after `withdrawal_epoch` was passed.</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/fadbcfd800cf32a02e0f8e026aab6c5ad453d99">https://github.com/rocket-pool/rocketpool/commit/fadbcfd800cf32a02e0f8e026aab6c5ad453d99</a></p> <p>We refer to our general resolution comment for this contract.</p>
<b>Resolution 2 / Follow-Up</b>	<p>Resolved, the design has been refactored and reward distribution is now locked as long as exiting validators have exited, which includes periods before <code>withdrawable_epoch</code> has ever been reached.</p> <p>We however consider this issue as resolved as now the active attack path is essentially mitigated and this design is enforced, removing any ambiguous state.</p>

<b>Issue_56</b>	Reward disruption in case of exiting process from dissolved validator [FOLLOWUP]
<b>Severity</b>	Low
<b>Description</b>	<p>The reward pausing process via numExitingValidators has the intention to prevent reward distribution as long as validators are exiting because it is expected that ETH is being received and this ETH should not be treated as rewards.</p> <p>With a dissolved validator that has matching credentials it is also expected to receive ETH, however, this ETH amount is not being refunded to the NO because of a fix which fully disables the refund process for dissolved validators. Instead, these funds are simply allocated as rewards (which is acceptable to do).</p> <p>A problem occurs now that despite these funds not being refunded, the reward distribution mechanism remains disabled while it isn't really required to disable it in this specific scenario.</p>
<b>Recommendations</b>	Consider acknowledging this issue, it is not recommended to change the control-flow.
<b>Comments / Resolution</b>	

Issue_57	Average <code>capitalRatio</code> can be updated even if <code>stake</code> has not happened [FOLLOWUP]
Severity	Low
Description	<p>Currently, <code>assignFunds</code> increases <code>nodeBond</code> and <code>userCapital</code> without snapshotting it. This is in itself accurate as the validator is not yet producing any rewards, hence it should not yet impact the ratio.</p> <p>However, if another interaction happens before the <code>stake</code> call, e.g. <code>notifyExit</code>, it will now call <code>_snapshotCapitalRatio</code> which incorporates the adjusted <code>nodeBond</code> / <code>userCapital</code> from the assignment.</p> <p>Effectively, assigned but not yet staked validators will now influence the <code>capitalRatio</code>.</p>
Recommendations	<p>In an effort to fix one of the most severe DoS issues, we consider a recommendation of calling updating the <code>capitalRatio</code> already during the <code>assignFunds</code> function for the very first assignment. While this is not necessarily accurate (as above described), it seems to be a low-intrusive fix for an impactful issue.</p> <p>However, all side-effects from that behavior need to be carefully analyzed.</p>
Comments / Resolution	<p>Acknowledged. The <code>assignFunds</code> function now snapshots the current <code>capitalRatio</code> for the very first execution (and potential subsequent execution if the enqueued validator is dequeued again). This fix was made to resolve another issue and results in this issue becoming acknowledged as now even without any subsequent state transition, ratio is being pushed without <code>stake</code> being called (which is the essence of this issue).</p> <p>The core principle remains the same for subsequent enqueueing as another transition then also pushes the new ratio if the validator has not been staked yet.</p>

<b>Issue_58</b>	Average <code>capitalRatio</code> update during <code>stake</code> is not fully accurate [FOLLOWUP]
<b>Severity</b>	Low
<b>Description</b>	Whenever <code>stake</code> is called, it will now incorporate the adjusted <code>nodeBond</code> and <code>userCapital</code> from the prior assignment. This is not 100% accurate as after <code>stake</code> , the validator does not get immediately activated and hence influences the <code>capitalRatio</code> without yet providing rewards.
<b>Recommendations</b>	In an effort to fix one of the most severe DoS issues, we consider a recommendation of calling updating the <code>capitalRatio</code> already during the <code>assignFunds</code> function for the very first assignment. While this is not necessarily accurate (as above described), it seems to be a low-intrusive fix for an impactful issue.  However, all side-effects from that behavior need to be carefully analyzed.
<b>Comments / Resolution</b>	Acknowledged, the principle remains the same as the validator is not directly considered as staked on the BC right after the <code>stake</code> call but the ratio is updated which is slightly desynced on a timing scale.

<b>Issue_59</b>	Final rewards during exit are allocated to the user side [FOLLOWUP]
<b>Severity</b>	<b>Low</b>
<b>Description</b>	<p>Currently, the <code>notifyFinalBalance</code> accepts <code>_amountInGwei</code> which represents final withdrawal balance including potential rewards.</p> <p>Any rewards should usually be distributed according to the distribution pattern but in this specific scenario these are simply allocated to the user side:</p> <pre><code>uint256 toUser = withdrawalBalance - toNode;</code></pre>
<b>Recommendations</b>	Consider acknowledging this issue.
<b>Comments / Resolution</b>	Acknowledged.

Issue_60	Exiting validators will prevent reward distribution for current average ratio [FOLLOWUP]
Severity	Low
Description	<p>RocketMegapoolDelegate distributes rewards via <code>distribute()</code> / <code>_distributeAmount()</code> using <code>calculateRewards()</code>, which derives the protocol/user/node split from <code>RocketNetworkRevenues</code>.</p> <p>The split depends on the node's <b>capital ratio</b> over time. Capital ratio snapshots are updated through <code>_snapshotCapitalRatio()</code>, which is invoked in multiple lifecycle transitions (e.g., <code>stake</code>, <code>reduceBond</code>, <code>dissolveValidator</code>, <code>_notifyFinalBalance</code>).</p> <p>To reduce retroactive re-pricing, <code>_snapshotCapitalRatio()</code> first attempts to distribute pending rewards <b>before</b> updating the capital ratio, but only when there are no <code>exiting</code> or <code>locked</code> validators.</p> <p>When <code>numExitingValidators &gt; 0</code> and/or <code>numLockedValidators &gt; 0</code>, <code>_snapshotCapitalRatio()</code> skips <code>_distributeAmount(getPendingRewards())</code>, but it still snapshots the new capital ratio via <code>RocketNetworkRevenues.setNodeCapitalRatio(nodeAddress, capitalRatio)</code> (if <code>nodeBond &gt; 0</code>).</p> <p>Because <code>lastDistributionTime</code> is only advanced when <code>_distributeAmount()</code> runs, the “distribution period” does not reset when distribution is blocked. When distribution becomes possible later, <code>calculateRewards()</code> computes the split using:</p> <ul style="list-style-type: none"> <li>• the total pending rewards amount at that later time, and</li> <li>• the <b>time-weighted average capital ratio</b> since <code>lastDistributionTime</code>.</li> </ul> <p>The borrowed portion calculation is non-linear in the ratio (<code>amount - amount / averageCapitalRatio</code>). Therefore, updating capital ratio snapshots while blocking distribution can cause rewards accrued prior to the ratio change to be split differently than they would have been if distribution had been executed immediately before the</p>

	<p>snapshot update.</p> <p>A privileged actor [trusted node] can also force a <code>locked</code> state via <code>RocketMegapoolManager.challengeExit</code>, preventing distribution while allowing capital ratio snapshots to continue, thereby amplifying this effect during periods where the node capital ratio changes [e.g., around <code>reduceBond</code>].</p>
<b>Recommendations</b>	Consider accepting this behavior.
<b>Comments / Resolution</b>	Acknowledged.

<b>Issue_61</b>	<code>reduceBond</code> can be called while no validator is staked [FOLLOWUP]
<b>Severity</b>	Low
<b>Description</b>	<p>Currently, the <code>reduceBond</code> function can be called already after assignment has happened while no validator is staked. This does not seem to be a valid state and may serve as precondition for advanced exploits.</p> <p>Most of the time, exploits happen due to arbitrary user inputs or users invoking functions which are not meant to be invoked by users, one can argue that a large user flexibility is a great seed for exploits. Therefore, at BailSec, we are of the opinion that codebases should never provide more user flexibility than necessary during the normal business logic.</p>
<b>Recommendations</b>	Consider thinking about further hardening the <code>reduceBond</code> function.
<b>Comments / Resolution</b>	Resolved.

Issue_62	Temporary <code>dequeue</code> revert in edge-case [FOLLOWUP]
Severity	Low
Description	<p>Currently it is possible that <code>dequeue</code> reverts due to an underflow of <code>userCapital</code> via an edge-case of <code>reduce.bond</code> decrease with validator finalization and queued validators:</p> <p>Alice has 3 staked validators [4,4,4]</p> <ul style="list-style-type: none"> <li>- <code>nodeBond</code> = 12</li> <li>- <code>userCapital</code> = 84</li> </ul> <p>Alice queues validator 4</p> <ul style="list-style-type: none"> <li>- <code>nodeBond</code> = 12</li> <li>- <code>userCapital</code> = 84</li> <li>- <code>nodeQueuedBond</code> = 4</li> <li>- <code>userQueuedCapital</code> = 28</li> </ul> <p><code>reduced.bond</code> is set to 2</p> <p>Validator 1 exits</p> <ul style="list-style-type: none"> <li>- <code>effectiveBond</code> = 16</li> <li>- <code>requiredBond</code> = 8</li> <li>- <code>_nodeShare</code> = 8</li> <li>- <code>_userShare</code> = 24</li> <li>- <code>nodeBond</code> = 4</li> <li>- <code>userCapital</code> = 60</li> <li>- <code>nodeQueuedBond</code> = 4</li> </ul> <p>The system is now collateralized for 3 validators; 2 are staked; 1 is queued</p> <p>Attempt to dequeue validator 4 reverts</p> <ul style="list-style-type: none"> <li>- <code>nodeQueuedBond</code> = 0</li> <li>- <code>nodeBond</code> = 4</li> <li>- <code>newBondRequirement</code> = 6</li> <li>- <code>require[nodeBond + nodeQueuedBond &gt;= newBondRequirement, "Bond requirement not met"];</code> <ul style="list-style-type: none"> <li>- <code>revert</code></li> </ul> </li> </ul>

The root-cause seems to be that the exit of validator 1 already paid out the full excess, while now the `dequeue` decreases `nodeQueuedBond` by the initial queued amount which means it would become undercollateralized

That seems to be one of the reasons why `reduceBond` is not callable when there are queued validators, it just did not incorporate the fact that a validator exit does under the hood the exact same logic as `reduceBond`.

The main impact here is that a NO cannot `dequeue` a validator but instead needs to assign and stake/dissolve a validator, followed by the normal exit flow

Another higher tier impact is if `getAssignDepositsEnabled() == false`, then normally `dequeue` would be the way to exit but it's broken, thus the validator remains locked in queued state.

A similar issue exists within `dissolveValidator`:

`reduced.bond = 1`

Queue and assign validator 1

- `nodeBond = 4`
- `userCapital = 28`

Queue validator 2, not assign it yet

- `nodeBond = 4`
- `userCapital = 28`
- `nodeQueuedBond = 1`
- `userQueuedCapital = 31`

Attempt to dissolve validator 1

- `requiredBond = 4`
- `effectiveBond = 5`
- `_nodeShare = 1`
- `_userShare = 31`

	<p>Attempting to decrease <code>userCapital</code> by 31 ETH will underflow as long as the queued validator is not assigned. It's more dramatic if assignments are prevented or for a longer time there are no funds in the megapool. This means one can purposefully DoS <code>dissolveValidator</code> in this specific scenario, but it does not seem that there will be any harm here because <code>stake</code> cannot be called either, it will just lock funds until validator 2 is assigned and allows us to dissolve validator 1</p> <p>The revert is technically caused by subtracting <code>userShare</code> from <code>userCapital</code> even though <code>userShare</code> is computed using an “active count” / “effective bond” that includes queued validators, while the corresponding “user side” for queued validators sits in <code>userQueuedCapital</code> and is not available to be debited in <code>dissolveValidator</code>.</p>
<b>Recommendations</b>	<p>A fix for this issue would be to loosen up the strictness check within <code>dequeue</code> but this means there is a risk introduction for other issues.</p> <p>Since this issue is only temporary and will get mitigated as soon as assignment has happened, it can be a reasonable solution to just acknowledge it.</p> <p>For the dissolve issue, it would require fundamentally changing the way how <code>nodeQueuedBond</code> is incorporated into the <code>effectiveBond</code> calculation and further capital dispersal.</p> <p>We do not recommend changing this mechanism as this will certainly introduce a re-audit requirement.</p>
<b>Comments / Resolution</b>	Acknowledged.

Issue_63	<code>distributeAmount</code> updates <code>lastDistributionTime</code> even if there are no rewards [FOLLOWUP]
Severity	Low
Description	The <code>distributeAmount</code> function is callable even if there are zero rewards and will advance <code>lastDistributionTime</code> . This practice allows for distorting the average <code>capitalRatio</code> .
Recommendations	<p>Changing this behavior will introduce a precondition for a critical vulnerability., as now an attacker could enter a state where there is a staked validator (user-provided ETH) while <code>lastDistributionTime = 0</code>, which then means the attacker can exploit the state to distribute exited 32 ETH fully towards NO (due to <code>lastDistributionTime = 0</code>).</p> <p>We do not recommend changing this.</p>
Comments / Resolution	<p>Acknowledged. Please note that it is also possible to call this function while there are no staked validators, in the scenario of assignment + dissolve it will set <code>lastDistributionTime &gt; 0</code> while there are no staked validators which means <code>distribute</code> can be called frequently.</p> <p>This means the following invariant is violated:</p> <p><i>// Prevent calls before a megapool's first validator has been staked"</i></p>

<b>Issue_64</b>	Megapool debt can stay permanently elevated in case the last validator's exit did not offset all debt
<b>Severity</b>	Low
<b>Description</b>	Currently, there is no enforcement that debt is fully resolved if the last validator is withdrawn. This process is natural as it is not possible to request funds from a validator if he is more in debt than the pointed collateral. Insolvency will occur in that scenario.
<b>Recommendations</b>	Consider acknowledging this issue.
<b>Comments / Resolution</b>	Acknowledged.

<b>Issue_65</b>	Lack of reset for <code>lastAssignmentTime</code> during <code>notifyFinalBalance</code> [FOLLOWUP]
<b>Severity</b>	Informational
<b>Description</b>	The <code>lastAssignmentTime</code> is not reset during <code>notifyFinalBalance</code> . This is only an informational issue as this value is never used for any accounting purpose
<b>Recommendations</b>	Consider acknowledging this issue or resetting this value.
<b>Comments / Resolution</b>	Acknowledged.

<b>Issue_66</b>	Challenge late notification window ignores dropped blocks, penalizing nodes during Ethereum instability [FOLLOWUP]
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>The challenge system uses timestamp-based deadlines for proof submission, but actual submission requires block inclusion. During Ethereum block producer issues (missed proposals, attestation problems), timestamps advance normally (12s per slot) while transaction inclusion opportunities are reduced.</p> <p>Nodes may be penalized not due to misbehavior but inability to land transactions during degraded block production. The challenge window doesn't account for reduced throughput - a node could have the same timestamp window but significantly fewer blocks to submit their proof.</p>
<b>Recommendations</b>	Consider extending grace periods to account for block production variance or adding a mechanism to pause/extend challenges during detected network degradation.
<b>Comments / Resolution</b>	Acknowledged.

<b>Issue_67</b>	Old averageCapitalRatio is being used if all validators have exited [FOLLOWUP]
<b>Severity</b>	<b>Low</b>
<b>Description</b>	<p>Whenever all validators have exited, <code>nodeBond = 0</code> and <code>userCapital = 0</code> which means no further snapshots are posted.</p> <p>If now there are some funds being received in the megapool, these funds are distributed using the old <code>averageCapitalRatio</code>.</p>
<b>Recommendations</b>	Consider acknowledging this issue.
<b>Comments / Resolution</b>	

<b>Issue_68</b>	Redundant <code>nodeQueuedBond</code> incorporation into <code>reduceBond</code> [FOLLOWUP]
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>The <code>nodeQueuedBond</code> value is incorporated into a check within <code>reduceBond</code> while it is actually required to be zero:</p> <pre><code>require[nodeQueuedBond == 0, "Cannot reduce bond with queued validators"]; // Check bond requirements RocketNodeDepositInterface rocketNodeDeposit = _getRocketNodeDeposit(); uint256 newBondRequirement = rocketNodeDeposit.getBondRequirement[getActiveValidatorCount()]; uint256 effectiveBond = nodeBond + nodeQueuedBond;</code></pre> <p>This is redundant.</p>
<b>Recommendations</b>	Consider removing <code>+ nodeQueuedBond</code>
<b>Comments / Resolution</b>	Resolved.

<b>Issue_69</b>	<code>lockedSlot</code> is never reset
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	The <code>lockedSlot</code> which is set during <code>challengeExit</code> is never reset and remains stale. This can confuse third-party reviewers.
<b>Recommendations</b>	Consider resetting this value.
<b>Comments / Resolution</b>	Resolved.

<b>Issue_70</b>	Insufficient storage transitions for <code>ValidatorInfo</code>
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>Currently, there are multiple inconsistencies for <code>ValidatorInfo</code> in case of <code>dequeue</code> call, such as:</p> <ul style="list-style-type: none"> <li>- <code>prestakeSignature</code> not deleted</li> <li>- <code>pubkey</code> not deleted</li> <li>- <code>ValidatorInfo</code> values not adjusted</li> <li>- ...</li> </ul>
<b>Recommendations</b>	Consider thinking about whether it is necessary to adjust these values.
<b>Comments / Resolution</b>	<p>Open. The Rocketpool team provided the following comment:</p> <p>Fixed by zeroing the relevant storage variables.</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/598d3f87e5f4f88449e03cc3ad491b2ea61c83a9">https://github.com/rocket-pool/rocketpool/commit/598d3f87e5f4f88449e03cc3ad491b2ea61c83a9</a></p> <p>And deleting '<code>prestakeSignature</code>' on '<code>dequeue</code>'.</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/dafdf2438c971f49a327b0858a2f0ccb364b8a77b">https://github.com/rocket-pool/rocketpool/commit/dafdf2438c971f49a327b0858a2f0ccb364b8a77b</a></p> <p>We refer to our general resolution comment for this contract.</p>
<b>Resolution 2 / Follow-Up</b>	Resolved.

## RocketMegapoolDelegateBase

The `RocketMegapoolDelegateBase` contract is the base contract which inherits the `RocketMegapoolDelegate` contract and these two contracts together form the implementation contract which is used by the `RocketMegapoolProxy` contract for delegatecalls.

It exposes all important modifiers and also the `deprecate` logic which is consulted by the `RocketMegapoolProxy` contract.

### Privileged Functions

- `deprecate`

Issue_71	Lack of idempotency within <code>deprecate</code> will result in constant extension of <code>expirationBlock</code>
Severity	High
Description	<p>Currently, the <code>deprecate</code> function is callable multiple times and each time it is called, it will re-shift the <code>expirationBlock</code>.</p> <p>Since this function is triggered here:</p> <pre><code>if [expiry == 0    block.number &lt; expiry] {     // This delegate is still "in-use" so set the expiry block into     // the future     delegate.deprecate();     deprecatedOne = true; }</code></pre> <p>We can see that <code>deprecate</code> is called if <code>block.number &lt; expiry</code>. In the scenario where <code>_upgradeDelegate</code> is called multiple times within the <code>expiry</code> period, it will always re-shift the <code>expirationBlock</code>. It has to be noted that the <code>upgradeBuffer</code> is 860000 blocks, which translates to 120 days. Therefore, if called the second time within 120 days, it will re-shift the old delegate, allowing users to still use it. This is severe as it may happen that a constant re-shift is applied.</p>
Recommendations	<p>Consider making the <code>deprecate</code> function idempotent.</p> <p>Alternatively, the <code>RocketMegapoolFactory._upgradeDelegate</code> function can be adjusted to not call <code>delegate.deprecate</code> in case the <code>expirationBlock</code> has already expired.</p>
Comments / Resolution	<p>Acknowledged. The Rocketpool team considers this as a non-issue with the following comment:</p> <p>This is intentional. Because when a delegate expires it is essentially forced onto the latest or the old one stops working. A malicious oDAO could upgrade the delegate to a non-malicious one. Then right before all the non-upgraded delegates expire, upgrade to a malicious one. This would give NOs no time to react as they are almost immediately forced to the latest malicious one.</p>

To prevent this type of attack, any time a new delegate is upgraded, all non-expired delegates have their expiration date reset to a future time.

**While this explanation indeed makes sense to cover the edge-case of malicious oDAO, we are still of the opinion that it is a severe issue that users may constantly use the old delegate in the described edge-case.**

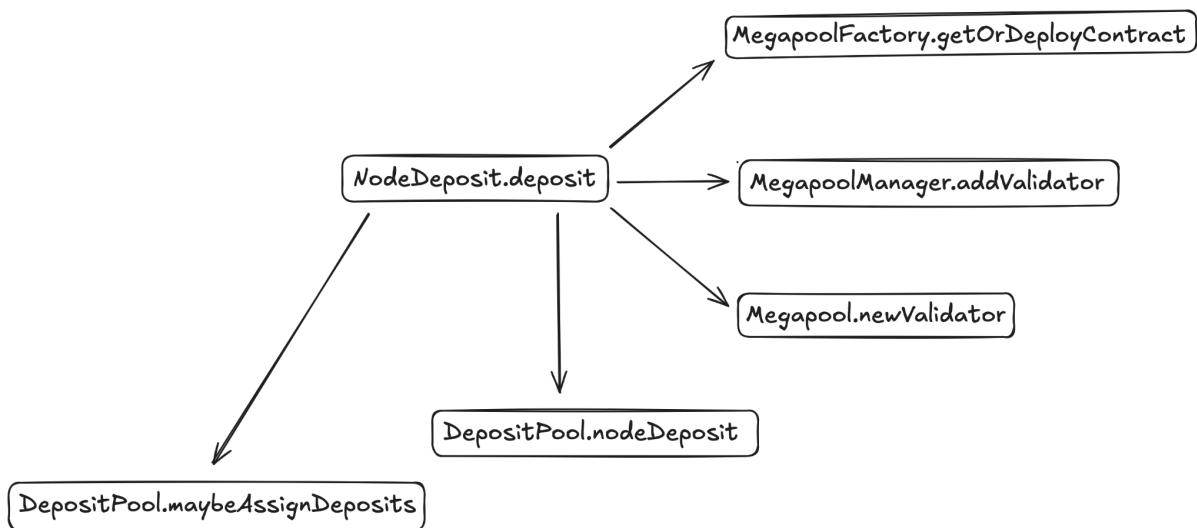
Issue_72	User-flexibility for delegate usage can become problematic
Severity	Low
Description	<p>Currently, users are not forced to use a new delegate once it is set and can still use the old implementation address. This will become a severe issue if there are unidentified critical issues in the delegate contract as now an update does not prevent this from getting exploited.</p> <p>The same counts for minipools as well.</p>
Recommendations	We recommend reconsidering this design choice.
Comments / Resolution	<p>Acknowledged. The Rocketpool team considers this as a non-issue with the following comment:</p> <p>Users are forced onto a new delegate after 120 days. This is an intentional design choice to offer NOs a level of protection against a malicious oDAO upgrading to a contract which could steal their ETH deposit.</p>

Issue_73	block time change can result in unexpected shift of <code>upgradeBuffer</code>
Severity	Informational
Description	<p>Currently, <code>upgradeBuffer</code> relates to <code>block.number</code>. However, in the case of future mainnet upgrades, this will become an issue because the actual expected time does not match the newly expected time. Furthermore, current running expiries can now expire later/sooner than expected.</p>
Recommendations	Consider using the <code>block.timestamp</code> and seconds for this purpose.
Comments / Resolution	Resolved.

## RocketMegapoolFactory

The `RocketMegapoolFactory` contract is responsible for the Megapool deployment and the upgrade mechanism.

### Appendix: Deployment Flow



### Appendix: Upgrade Delegate

The `_upgradeDelegate` function is responsible for upgrading to the newest delegate and keeping track of head and tail. Head is related to the index of the oldest but not yet expired delegate contract and tail is related to the index of the newest added delegate. On each `_upgradeDelegate` call, a loop is executed which attempts to sort out all expired delegates and thus advances the head to reflect the expiration.

It has to be noted that the `upgradeDelegate` function relies on the `onlyLatestNetworkContract` modifier and is expected to be used in the future.

#### Core Invariants:

INV 1: An external deprecate call must be executed to any delegate contract that has no expiration block

INV 2: deployContract must revert if Megapool is already existing



INV 3: Initially, uint for metaKey must be zero

INV 4: tail must increase whenever new delegate contract is added

INV 5: head must increase if a delegate is expired

#### Privileged Functions

- initialise
- deployContract
- getOrDeployContract
- upgradeDelegate

No issues found.

## RocketMegapoolManager

The [RocketMegapoolManager](#) contract mainly serves as a safeguard for certain state transitions on Megapools, such as:

- [stake](#): Ensures that withdrawal credentials match with the Megapool address
- [dissolve](#): Ensures dissolve can only be called if credentials do not match
- [notifyNotExit](#): Ensures the validator state is valid and has not exited
- [notifyExit](#): Ensures the validator has a set withdrawable\_epoch
- [notifyFinalBalance](#): Ensures a withdrawal has indeed happened

For this purpose, the [BeaconStateVerifier](#) contract is leveraged which validates the validator state and the withdrawal state against the consensus states.

It furthermore implements a mechanism to challenge validators in the scenario where they are exited but have not notified the Megapool for that state. This is to ensure rewards cannot be distributed when the nearest withdrawable\_epoch is reached, while a validator has exited but not set withdrawableEpoch. This is executable by any trusted node.

Core Invariants:

INV 1: challengeExit cannot be called twice subsequently by the same trusted node

Privileged Functions

- [challengeExit](#)
- [addValidator](#)

Issue_74	Lack of exit check during stake
Severity	High
Description	<p>Within the <code>MegapoolDelegate</code> contract, we have added an issue that exited validators can be added to a Megapool. There are multiple side-effects from that issue and for these we simply refer to the corresponding issue.</p> <p>It has to be mentioned that even if such an exit check would be incorporated within the <code>MegapoolDelegate.newValidator</code> function, it would still be possible to then exit before assignment/stake is called to then produce the same faulty state where it is possible to manipulate the exchange rate due to a reward skim and distribution while a validator is not marked as exited and thus double counting of userCapital and the increased ETH balance within the RETH contract into the network eth balance.</p>
Recommendations	<p>Consider implementing a check for <code>withdrawable_epoch</code> of a validator during the <code>stake</code> function. It has to be mentioned that this only partially mitigates the root-cause as it is theoretically possible to provide a proof which is up to 27 hours old.</p> <p>Further mechanisms need to be developed to prevent this issue from happening. A potential idea would be to implement a scrub-like period which allows the DAO to interfere in such situations.</p>
Comments / Resolution	<p>Open. The Rocketpool team provided the following comment:</p> <p>Added check that `withdrawable_epoch` and `exit_epoch` are set to the correct value of `FAR_FUTURE_EPOCH` on `stake`. Also check `slashed` for good measure.</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/fadbc800cf832a02e0f8e026aab6c5ad453d992">https://github.com/rocket-pool/rocketpool/commit/fadbc800cf832a02e0f8e026aab6c5ad453d992</a></p> <p>Note: in the above commit we were checking `effective_balance` was 1 ETH but this turned out to be an issue as this can be modified</p>

	<p>by donating to the prestate validator. So we replaced this with a check that `effective_balance` &lt; 32 and further added checks for `activation_epoch` and `activation_eligibility_epoch`.</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/b45f9330982137c55d6a72d87294bf675524e">https://github.com/rocket-pool/rocketpool/commit/b45f9330982137c55d6a72d87294bf675524e</a></p> <p>This fix needs to be reviewed with the full exit lifecycle in context.</p>
<b>Resolution 2/ Follow-Up</b>	Resolved, multiple additional safeguards within the <code>stake</code> function have been implemented.

<b>Issue_75</b>	DoS due to permanent consumption of <code>pubkey</code> [FOLLOWUP]
<b>Severity</b>	Medium
<b>Description</b>	<p>Currently, anyone can use a <code>pubkey</code> to enqueue a validator just then to dequeue it again. This practice will mark the <code>pubkey</code> as consumed and will prevent it from being reused.</p> <p>This can become an annoying attack-vector if legitimate users' pubkeys are consumed that way.</p>
<b>Recommendations</b>	Consider implementing a <code>dequeue</code> fee to prevent such an attack,
<b>Comments / Resolution</b>	Resolved.

<b>Issue_76</b>	Loose sanity checks for input parameters
<b>Severity</b>	<b>Low</b>
<b>Description</b>	<p>Throughout the contract, there are multiple functions which do not force-validate the Megapool address, allowing to pass any arbitrary Megapool address and executing external calls outgoing to the said.</p> <p>Most of the time, exploits happen due to arbitrary user inputs or users invoking functions which are not meant to be invoked by users, one can argue that a large user flexibility is a great seed for exploits. Therefore, at BailSec, we are of the opinion that codebases should never provide more user flexibility than necessary during the normal business logic.</p>
<b>Recommendations</b>	Consider checking the Megapool address against the deployed Megapools: <a href="#">MegapoolFactory.getMegapoolDeployed</a> .
<b>Comments / Resolution</b>	Resolved in the following commit: <a href="https://github.com/rocket-pool/rocketpool/commit/62345f8279fb154c1fa848df4ebae842a2da24b">https://github.com/rocket-pool/rocketpool/commit/62345f8279fb154c1fa848df4ebae842a2da24b</a>

<b>Issue_77</b>	Mismatch between NATSPEC and code
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	The challengeExit function mentions that it is possible to challenge up to 50 validators. However totalChallenges is required to be < 50, hence only allowing to challenge up to 49 validators
<b>Recommendations</b>	Consider removing the NATSPEC or changing the validation to <= 50
<b>Comments / Resolution</b>	Resolved in the following commit: <a href="https://github.com/rocket-pool/rocketpool/commit/bbf84c544fe68fb33058d8e821197d77006b4cc3">https://github.com/rocket-pool/rocketpool/commit/bbf84c544fe68fb33058d8e821197d77006b4cc3</a>

<b>Issue_78</b>	Stale Proof allows staking to slashed validator [FOLLOWUP]
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>The 1-hour <code>slotRecencyThreshold</code> allows proofs that are significantly stale.</p> <p>All validator state checks verify the state at time of proof, not the current beacon chain state. An attacker could: [1] Create pre-stake validator, [2] Externally deposit 31 ETH to bypass protocol, [3] Get validator activated and slashed within the 1-hour window, [4] Submit old proof showing compliant pre-stake state, [5] Protocol deposits 31 ETH to now-slashed validator.</p> <p>The impact of this issue is limited due to the slashing cool down periods on the beacon chain. However, capital allocation still occurs and the now slashed validator holds user capital for excessive durations.</p>
<b>Recommendations</b>	We recommend acknowledging this issue as there is no harm and this is a simple flexibility question only. This is furthermore a design constraint by the recency mechanism.
<b>Comments / Resolution</b>	Acknowledged.

## RocketMegapoolPenalties

The [RocketMegapoolPenalties](#) contract exposes the interface for trusted nodes to penalize Megapools. This mechanism works the same way as with most DAO related actions - by simply checking whether a quorum has been reached and then a subsequent execution.

The quorum for penalty is > 50% of the member count for trusted nodes. A safeguard has been implemented to ensure that the penalty amount cannot exceed a certain threshold.

### Appendix: Maximum Penalty

The maximum penalty logic is a safeguard which clamps the penalty amount which can be applied within the range of 50\_400 blocks. It is set by default to 612 ETH and this essentially means that within a period of 50\_400 blocks, no more than 612 ETH penalty can be applied. This is not time-linear based but checkpoint based. This means if we start a period at block = 0 and at block = 10\_000 a penalty of 500 ETH is applied, it is only allowed to apply 112 more ETH as penalty until before block 60\_400, as then at block 60\_400, the lookback period of 50\_400 does already include the penalty value at 10\_000 which is 500 and thus currentTotal = 0. The penalty is thus “reset”.

Illustrated:

- `penalty[10_000] = 500`

If there is no further penalty and we fetch the currentTotal at any point after/equal 10\_400, we will have a currentTotal of 50\_400:

`currentRunningTotal[60_000] = 500`

`earlierRunningTotal[10_000] = 0`

`> currentTotal = 500`

Once we have reached 60\_400:

`currentRunningTotal[60_400] = 500`

`earlierRunningTotal[10_000] = 500`

`> currentTotal = 0`

This is done by leveraging the **NetworkSnapshots** contract

#### Core Invariants:

INV 1: penalise cannot be called twice with the same megapool/block/amount parameters by the same node

INV 2: During applyPenalty, \_amount cannot be larger than currentMaxPenalty

INV 3: Within penaltyMaximumPeriod, it is not allowed to aggregate penalty more than maximumETHPenalty

#### Privileged Functions

- penalize

<b>Issue_79</b>	Missing <code>amount &lt;= getMaximumETHPenalty</code> check will make submissions void
<b>Severity</b>	Low
<b>Description</b>	<p>Currently, there is no such check which ensures that the actual amount value is indeed within the allowed range. The creation of such a scenario is therefore theoretically possible but can then never be executed.</p> <p>Please note that this issue is also present within the NetworkPenalties contract, as these share the same logic around the running total.</p>
<b>Recommendations</b>	Consider implementing such a check.
<b>Comments / Resolution</b>	Resolved in the following commit: <a href="https://github.com/rocket-pool/rocketpool/commit/f9bea5de1978a4623fefafa86ceae26f53473f8d5a">https://github.com/rocket-pool/rocketpool/commit/f9bea5de1978a4623fefafa86ceae26f53473f8d5a</a>

Issue_80	Usage of <code>block.number</code> is suboptimal
Severity	Informational
Description	<p>The usage of <code>block.number</code> is suboptimal as it can happen that for example due to mainnet upgrades, the block duration time shifts, which will impact the penalty period.</p> <p>Please note that this issue is also present within the NetworkPenalties contract, as these share the same logic around the running total.</p>
Recommendations	<p>We do not recommend a change in an effort to keep the code unchanged. <b>And to not conflict with potential broken states post-upgrade.</b></p> <p>However, if something like that arises, we recommend recalculating the penalty and executing a governance vote to make sure the penalty aligns with the desired outcome.</p>
Comments / Resolution	<p>Open. This has been resolved by switching from <code>block.number</code> to <code>block.timestamp</code>, including using a new RocketNetworkSnapshotsTime contract. While there are no obvious issues, we are the opinion that this requires a careful review, including the new contract.</p>
Resolution 2 / Follow-Up	Resolved, <code>block.number</code> logic has been replaced with timestamps and this does not expose a problem because this contract has no historical data (was not existing pre-upgrade).

## RocketMegapoolProxy

The `RocketMegapoolProxy` contract is the unique implementation address which is used by the `Clones.cloneDeterministic` function. It acts as a proxy behind a proxy. Under the hood, the execution flow for a Megapool looks as follows:



Essentially, when a user triggers the initial call, this will execute a `delegatecall` to the `MegapoolProxy`, the `MegapoolProxy` will then fetch the correct `MegapoolDelegate` based on its internal mechanism. That can be either always the current delegate contract if `useLatestDelegate = true` or simply the latest determined `rocketMegapoolDelegate` contract.

### Core Invariants:

INV 1: initialise can only be called once

INV 2: fallback must revert in case `useLatestDelegate` is false and the delegate contract is expired

INV 3: fallback must always fetch the current “`rocketMegapoolDelegate`” incase `useLatestDelegate` is true

INV 4: `setUseLatestDelegate` must update the delegate contract to the most recent one in case it is set to false.

### Privileged Functions

- `setUseLatestDelegate`

<b>Issue_81</b>	No automatic upgrade in case of expiry
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	In the scenario where the delegate has been expired, the fallback simply reverts and requires a manual call of <code>delegateUpgrade</code> , which is permissionless.  This is a small UX disadvantage-
<b>Recommendations</b>	Consider directly fetching and updating the delegate contract in case of a fallback call where the current delegate is expired.
<b>Comments / Resolution</b>	Resolved.

## RocketMegapoolStorageLayout

The [RocketMegapoolStorageLayout](#) contract represents the storage layout for a Megapool. It is inherited by the [MegapoolDelegateBase](#) contract.

### Privileged Functions

- none

No issues.

## Minipool

### Appendix: Minipool Lifecycle

After Saturn, it is no longer permitted to create new Minipools via a normal creation or via a vacant Minipool. However, Minipools are still expected to be in the queue and must be assigned normally. Moreover, vacant Minipools can still be promoted. Therefore, the standard Minipool cycle is still existent:

#### Minipool Assignment:

Below we will illustrate the standard Minipool assignment when a Minipool is successfully assigned and the scrub period passes successfully:

#### Minipool Promotion:

In that scenario, the Minipool is simply promoted after the `minipool.promotion.scrub.period` [3d]. The status will be set to Staking and the NO will receive a credit which is equivalent to the `userDepositBalance`.

## RocketMinipoolBase

The `RocketMinipoolBase` contract is the implementation contract which is deployed via the `Clones` pattern and it then further executes a delegatecall to the final implementation contract, the `RocketMinipoolDelegate` contract.

This contract functions via a fallback mechanism that selects the `delegateContract` and forwards delegatecalls to it.

**This contract has not been changed for the Saturn upgrade.**

### Appendix: Upgrade and Rollback

The node operator can call `delegateUpgrade()` to update the pointer to the latest delegate contract published by the network registry.

Additionally, the operator can rollback using `delegateRollback()` to revert to the previous delegate version, after which `rocketMinipoolDelegatePrev` is cleared. This means, it is only possible to roll back exactly one time and it is not possible to roll back to the state before the previous state.

When `useLatestDelegate` is true, all calls are always routed to the newest network delegate automatically; otherwise, they continue to use the stored version until explicitly upgraded.

There is currently no mechanism to prevent calls to an outdated `MinipoolDelegate` contract.

### Core Invariants:

INV 1: `delegateRollback` can only be called once after an upgrade

### Privileged Functions

- `delegateUpgrade`
- `delegateRollback`

<b>Issue_82</b>	Missing sanity checks allow for interacting with outdated implementation contract
<b>Severity</b>	Low
<b>Description</b>	The fallback function does not expose any sanity check where one could determine that an old delegate contract is expired. There may be valid scenarios in case of unexpected bugs.
<b>Recommendations</b>	The fix for this issue would be to implement such a check. However, since this contract is already live and minipools are not meant to receive a new delegate [?], it does not change anything.
<b>Comments / Resolution</b>	Acknowledged. There will be no new minipool delegates

<b>Issue_83</b>	Interaction possibility with three different implementation contracts within the same block
<b>Severity</b>	Low
<b>Description</b>	<p>There is a possibility of interacting with three different delegate contracts at the same time:</p> <p>The contract is in a state where it does not reference to the latest delegate contract and also has a valid <code>rocketMinipoolDelegatePrev</code>. This exposes increased user flexibility in to which delegate to choose for calls. Essentially it allows for three interactions:</p> <ul style="list-style-type: none"> <li>a) Interact with <code>rocketMinipoolDelegatePrev</code></li> <li>b) <code>setUseLatestDelegate</code> call</li> <li>c) Interact with current delegate</li> </ul>
<b>Recommendations</b>	We recommend reconsidering this design choice.
<b>Comments / Resolution</b>	Acknowledged.

<b>Issue_84</b>	Potential version skip allows <code>rocketMinipoolDelegatePrev</code> to be very old delegate
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>Currently, it is possible that <code>rocketMinipoolDelegatePrev</code> becomes a very old version.</p> <p>This is related to the fact that in case the newest version is not current +1, the <code>rocketMinipoolDelegatePrev</code> will become an older version than newest -1.</p>
<b>Recommendations</b>	Consider acknowledging that issue. It must be clear that the assumption should be that some users can literally interact with any delegate version.
<b>Comments / Resolution</b>	Acknowledged.

<b>Issue_85</b>	Potential issues due to rollback mechanism
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>Certain legacy Minipoools might expose unidentified issues which are not covered within this audit. The rollback mechanism may allow to roll back to these implementations and it is likely that there are currently Minipoools pointing to such implementation.</p> <p>This is an inherent risk.</p>
<b>Recommendations</b>	Consider removing this mechanism. However, since this code is already live and non-upgradeable, a fix does not have any impact.
<b>Comments / Resolution</b>	Acknowledged.

## RocketMinipoolBondReducer

The [RocketMinipoolBondReducer](#) contract was used pre-saturn and is no longer actively in use. While it is still deployed and upgraded, all important functionalities will now just revert. This is to ensure backwards compatibility.

This contract has been changed for the Saturn upgrade.

### Privileged Functions

- none

<b>Issue_86</b>	Bond reduction cannot be completed after upgrade
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	In the scenario where users create 16 ETH bond minipools just before the upgrade with the intention to decrease the bond later, this will not be possible anymore after saturn.
<b>Recommendations</b>	Consider communicating this early on to ensure users will not be surprised. Furthermore we recommend disabling the <a href="#">beginReduceBondAmount</a> function on the frontend with sufficient time before the upgrade.
<b>Comments / Resolution</b>	Acknowledged. The Rocketpool team mentioned this is intentional. The severity has been downgraded to informational.

## RocketMinipoolDelegate

The [RocketMinipoolDelegate](#) contract is the implementation contract which is used by the [RocketMinipoolDelegateBase](#) contract and handles all logic around shared staking. In the pre-Saturn version, users were able to create Minipools by providing an 8/16 ETH bond amount which is then paired with user funds and the results in one staked validator that is related to the Minipool contract. All different mechanics will be illustrated in the below appendices.

**This contract has not been changed for the Saturn upgrade**

### Appendix: Minipool States

A created Minipool can have the following different states:

**Initialised:** Minipool is created and configured [owner, fees set]; no user deposit assigned yet, and it can move to Prelaunch via deposit or prepareVacancy.

**Prelaunch:** User capital is assigned [or pool is marked vacant], scrub window runs; it can be staked [stake], promoted [vacant path], or aborted via dissolve/scrub.

**Staking:** Validator deposit has been sent to the Beacon Chain; the pool accrues/skim-distributes rewards and later performs full distribution/finalisation when sufficient funds are present.

**Dissolved:** Prelaunch was aborted [timeout/scrub] and user capital is recycled [non-vacant]; owner can withdraw residuals and close the minipool.

### Appendix: Scrub Period

After a Minipool has been created and was assigned with a 1 ETH prestate deposit, the Minipool is transitioned into the [Prestake](#) phase. In that phase, it is possible that trusted members vote for a Minipool to be scrubbed. This is for safety reasons in the scenario where the withdrawal credentials do not match and the subsequent stake call would then accidentally deposit the remaining 31 ETH to a validator with malicious credentials.

If a Minipool has been successfully scrubbed, it will get automatically dissolved either with a penalty of 2.4 ETH [in case of a normal Minipool creation] or without any penalty [in case of a vacant Minipool]. While it is now theoretically possible to fetch the validator state on the

beacon chain directly from the `BeaconRoots` contract [as the Megapool module is doing], this code was not updated for security reasons.

## Appendix: Finalization

After a Minipool has transitioned into the Stake phase, it is considered as active and any rewards which are received are considered as distributable via the `distributeBalance` function. This function assumes that the validator has exited if the contract balance is  $\geq 8$  ETH.

There are two scenarios how this is triggered.

### a) Owner calls `distributeBalance`:

It will simply calculate the `nodeShare` including potential rewards and penalty and deduct this from the balance which then determines the `userAmount`.

The `userAmount` is transferred directly to the `RETH` contract and the `nodeShare` is used to increase `nodeRefundBalance`. Within the same call, the Minipool will be finalized which means a transfer out of the `nodeRefundBalance` and accounting in the `MinipoolManager` contract.

### b) User calls `distributeBalance`

A user can only call `distributeBalance` with the precondition that the `beginUserDistributeTime` function has been triggered 90 days before but not later than 92 days. This flow will then simply follow the same mechanics above with the difference that the Minipool is not yet finalized. The finalization and payout of `nodeShare` can only be triggered by the owner.

## Appendix: Vacant Minipools

The vacant mechanism onboard an *already-running* validator into Rocket Pool without pulling user capital upfront.

Via `prepareVacancy`, the contract

- checks `currentBalance  $\geq$  launchAmount`
- sets `nodeDepositBalance = bond`
- a synthetic `userDepositBalance = launchAmount - bond`
- records `preMigrationBalance`
- flags `vacant = true`
- credits the node any pre-RP rewards (`nodeRefundBalance = currentBalance - launchAmount`)
- moves to Prelaunch

While vacant, refunds via `_refund()` are blocked and, if dissolved, the pool is simply removed from the vacant set [no user capital is recycled because none was drawn].

After the promotion scrub period, the node calls `promote()`:

- vacant is cleared
- status becomes Staking
- `userDepositAssignedTime` is set
- the node receives a deposit credit equal to `userDepositBalance`

Additionally, scrubbing a vacant pool does not apply the scrub penalty. The NO can then simply use the credit to create a Megapool validator or to mint RETH. In the minting RETH case, it can be simplified that a NO exchanges his user share of 24/16 ETH to the equivalent of RETH (after fee).

#### Appendix: Reward Calculation

Rewards are split by capital ratio with a commission: for a given balance, the contract computes total capital user + node; any excess over capital is “rewards.”

Below 8 ETH, the entire `_balance` is treated as rewards and the node gets:

```
> nodePortion = rewards * nodeCapital / (userCapital + nodeCapital)
```

plus a commission on the user’s portion:

```
> userPortion * nodeFee / 1e18.
```

At/above 8 ETH, `_calculateNodeShare` also returns principal: if `_balance > capital`, node gets `nodeCapital + rewards` share; if only `_balance > userCapital`, node gets `_balance - userCapital` (user gets their full principal first).

Finally, a penalty rate [from `RocketMinipoolPenalty`] reduces the computed node share. This penalty will be collected by slashing staked RPL.

#### Appendix: Validator Dissolve

A minipool can be dissolved either via the `dissolve` function or during scrubbing. That means, dissolving is only possible during the `Prestake` period, when 1 ETH is already deposited into the `CasperDeposit` contract.

**Dissolve via scrubbing:** In that scenario, the Minipool is marked as dissolved and 16/8 ETH + 2.4 ETH [penalty] is recycled into the `DepositPool`. This means, there will be a remaining of 31 -

16/8 - 2.4 ETH which can be withdrawn by the NO via the `distributeBalance` or `close` function.

**Dissolve without scrubbing:** Anyone can dissolve a non-scrubbed Minipool during the `Prestake` phase via the `dissolve` function [after the timeout period of 3d]. This will simply recycle the `userDepositBalance` of 16/24 ETH and mark the Minipool as dissolved. The Minipool owner can then withdraw funds via `distributeBalance` or `close`.

#### Appendix: `nodeFee`

The `nodeFee` value is determined during the Minipool initialization or during bond reduction by calling `NetworkFees.getNodeFee`. This follows a special algorithm to determine the `nodeFee` by the current demand. However, after Saturn the usage of this function is deprecated as no control-flow is invoking `getNodeFee`.

#### Core Invariants:

INV 1: A minipool must only be set to Staking via [stake](#) and [promote](#)

INV 2: Any staking trigger must call [incrementNodeStakingMinipoolCount](#)

INV 3: [During reward calculation](#), a part of the user portion is granted to node, based on `nodeFee`

INV 4: Penalty must be deducted from `nodeShare` during withdrawal

INV 5: [preDeposit](#) is never callable

INV 6: [userDeposit](#) is never callable

INV 7: `distributeBalance` must reset [userDistributeTime](#)

INV 8: `distributeBalance` is only callable [if Minipool is staking](#)

INV 9: [reduceBondAmount](#) is not callable anymore

INV 10: `nodeRewards` must contain `nodePortion` and [fee on userPortion](#)

INV 11: During `distributeSkimmedRewards`, `nodeShare` must [flow into nodeRefundBalance](#) and the leftover “`userShare`” must be [transferred to RETH](#)



INV 12: beginUserDistribute is only callable [if balance is >= 8 ETH](#)

INV 13: [distributeSkimmedRewards](#) can only be triggered if contract balance is < 8 ETH

INV 14: Within \_distributeBalance, if [balance < userCapital](#), all balance is distributed as userAmount

INV 15: Promoting only allowed after scrubPeriod has passed

### Privileged Functions

- preDeposit
- userDeposit
- prepareVacancy
- reduceBondAmount
- deposit
- voteScrub
- stake
- close
- finalise
- refund
- promote

Issue_87	Manipulation of ETH bonded by withdrawing bonded amount without finalizing the minipool
Severity	High
Description	<p>The minipool lifecycle in <code>RocketMinipoolDelegate</code> distinguishes two distribution regimes (threshold at 8 ETH):</p> <ul style="list-style-type: none"> <li>a) Full distribution (<math>\geq 8</math> ETH):           <ul style="list-style-type: none"> <li><code>distributeBalance[false]</code> calls <code>_distributeBalance[totalBalance]</code> where <code>totalBalance = address[this].balance - nodeRefundBalance</code>.</li> <li>i) If owner calls: it immediately <code>_finalise()</code>, which decrements the manager's staking count and performs refund/slash/leftover-to-rETH.</li> <li>ii) If non-owner calls: it does not finalise; it only enforces the time window (<code>beginUserDistribute</code> → <code>userDistributeAllowed</code>) and sets <code>userDistributed = true</code>.</li> </ul> </li>   <li>b) Reward distribution - unrelated for this attack</li> </ul> <p>After a validator voluntarily exits (<math>\approx 32</math> ETH landing on the minipool), a non-owner can run the full distribution path: <code>_distributeBalance</code> transfers the user amount to rETH and accrues the node amount to <code>nodeRefundBalance</code>.</p> <p>Because <code>totalBalance</code> is defined as the net of <code>nodeRefundBalance</code>, a subsequent call sees <code>totalBalance &lt; 8</code> ether, sending the owner into the partial branch where <code>_refund()</code> is reachable without <code>_finalise()</code>. As only <code>_finalise()</code> decrements the manager's staking count, the minipool remains active/staking while the node has already withdrawn their entire share.</p> <p>This has the effect that the <code>minipools.active.count</code> is not decreased while the bond was withdrawn, inflating the bonded ETH amount for minipoles.</p>

	<p>Please note that a similar attack is possible by using vacant minipool creation [see issue below]</p> <p><b>This issue is live in the current deployed version.</b></p>
<b>Recommendations</b>	<p>Usually, the recommendation would be to implement a different design to determine the bonded ETH amount and not using minipools.active.count minus the borrowed ETH amount.</p> <p>However, since this issue is in the live-codebase and historic state transitions were made using this assumption, it is not practicable anymore to implement a new design, because this would not be applied in hindsight.</p> <p>Therefore, this issue can be considered as partially resolved if the creation of new minipools is prevented.</p>
<b>Comments / Resolution</b>	Partially resolved. The Rocketpool team acted immediately as soon as we notified them of this issue and initiated a governance vote to disable the creation of new minipools. If there are existing vacant minipools, this will still expose a problem.

<b>Issue_88</b>	Vacant minipools can be used to inflate <code>minipools.active.count</code> and ETH bonded and prevent decrease of same
<b>Severity</b>	High
<b>Description</b>	<p>Currently, the creation of vacant minipools inflates <code>minipool.active.count</code> and thus inflates ETH bonded which then in turn can be exploited to bypass the clamping mechanism for the VP calculation.</p> <p>The root-cause lies within two things:</p> <ul style="list-style-type: none"> <li>a] Vacant minipools impact <code>minipools.active.count</code></li> <li>b) <code>MinipoolDelegate.close</code> can only be called by the owner.</li> </ul> <p>Please note that this is the second exploit which allows to maliciously increase ETH bonded for minipools and there are furthermore other exploit scenarios related to this root-cause. These are raised in separate issues.</p> <p><b>This issue is live in the current deployed version.</b></p>
<b>Recommendations</b>	<p>A full fix for this issue would be to refactor the ETH bonded mechanism for minipools.</p> <p>A partial fix would be to mark close as permissionless.</p> <p>And a practical fix would be to determine any NOs that have exploited this issue and penalize them, combined with disabling vacant minipools.</p>
<b>Comments / Resolution</b>	Resolved, the team followed the third recommendation. It has to be noted that vacant minipools must remain disabled for this issue to stay resolved.

Issue_89	Donation can force-slash Minipool owner
Severity	Medium
Description	<p>The public exit path for a staking minipool is using the contract balances as source of truth when the balance is <math>\geq 8</math> ETH, it is assumed that this is not a reward distribution but instead a validator withdrawal. This design choice is vulnerable to donation attacks that will result in slashing for the validator.</p> <p>A third party can donate exactly <math>\geq 8</math> ETH into a staking minipool without validator exit proceeds and then, through the user distribution path, force a slash of the node operator's RPL equal to <code>userCapital - donatedBalance</code> (capped by first-withdrawal logic). The attack does not require any action from the node operator.</p> <p>Illustrated:</p> <ul style="list-style-type: none"> <li>- Minipool is in Staking.</li> <li>- Let <code>userCapital = getUserDepositBalance()</code> [e.g., 24 ETH in a 8/24 configuration].</li> <li>- Assume <code>nodeRefundBalance == 0</code>.</li> <li>- Attacker forcibly sends 8 ETH to the minipool address.</li> <li>- Attacker calls <code>beginUserDistribute()</code></li> <li>- Preconditions pass because <code>status == Staking</code> and <code>address[this].balance - nodeRefundBalance == 8 ether</code>.</li> <li>- After the waiting period [e.g., ~90 days per settings], attacker calls <code>distributeBalance[_rewardsOnly = false]</code></li> <li>- Branch <code>totalBalance &gt;= 8 ether</code> is taken.</li> <li>- <code>_distributeBalance[totalBalance]</code> executes with <code>_balance = 8 ether</code>.</li> <li>- Since <code>_balance &lt; userCapital</code> and this is the first withdrawal [<code>withdrawalBlock == 0</code>], it sets <code>nodeSlashBalance = userCapital - 8 ether</code> [e.g., 16 ETH], pays <code>userAmount = 8 ETH</code> to <code>rocketTokenRETH</code>, and records <code>withdrawalBlock</code>.</li> <li>- Attacker (or anyone) calls <code>slash()</code>, burning/confiscating 16 ETH equivalent in RPL from the node [<code>rocketNodeStaking.slashRPL</code>].</li> </ul> <p>It is clear that the attacker loses 8 ETH in this step (depending on</p>

	how much RETH he owns, there is some partial redistribution back to the attacker) but it will essentially damage the NO with 16 ETH.
<b>Recommendations</b>	<p>To fix this issue, it is required to refactor the withdrawal flow. It should not rely on the contract balances as a source of truth.</p> <p>However, a full new audit for such a change would be required and given the current state of Rocketpool, it is questionable whether such a large change should be done.</p> <p>On top of that, the issue is only marked as medium severity because the code clamps the slashing balance within the NodeStaking contract, which allows a NO to bypass this attack if there is no RPL stake.</p>
<b>Comments / Resolution</b>	Acknowledged. This is a known limitation of the Minipool design.

<b>Issue_90</b>	Unfair reward distribution for period between [statusTime, promote]
<b>Severity</b>	Medium
<b>Description</b>	Whenever a minipool is marked as vacant, any balance > 32 ETH is allocated to <code>refundValue</code> . Afterwards, rewards from [statusTime; promote] are relayed to the minipool and then distributed [after promotion]. This is technically incorrect as any rewards are solely entitled to the validator until the promotion is executed.
<b>Recommendations</b>	Consider implementing a mechanism which correctly accounts for these rewards. It has to be noted that this requires a full re-audit of the vacancy/promotion control-flow.
<b>Comments / Resolution</b>	<p>Partially resolved, Bailsec reported another live issue during the audit which had the consequence to disable vacant minipool creation. It has to be noted that this is partially resolved because the chance exists that there have been vacant minipools which were not yet activated.</p> <p>It has to be noted that vacant minipools must remain disabled for this issue to stay partially resolved.</p>

<b>Issue_91</b>	Broken incentive model for fee recipient/coinbase address
<b>Severity</b>	Medium
<b>Description</b>	Currently, slashing is not applied on rewards [contrary to the megapool setup]. This design choice is suboptimal as now a validator can decide to simply forfeit his node bond and accept the slashing [and even never exit] while relaying all MEV rewards/fees to the own address instead of the distributor.
<b>Recommendations</b>	A fix for this issue is to refactor the slashing approach and to apply slashing on the reward amount of a NO. However, given that this is an intrusive upgrade and requires a full audit of the new control-flow, we recommend the Rocketpool team to further think about whether this issue should be mitigated.
<b>Comments / Resolution</b>	Acknowledged.

<b>Issue_92</b>	Side effects in <code>withdrawalBlock</code> assumptions
<b>Severity</b>	<b>Low</b>
<b>Description</b>	<p>The <code>withdrawalBlock</code> logic was implemented in scenarios where an exit with a shortfall is happening, followed by a reward distribution with <math>\geq 8</math> ETH.</p> <p>In itself, this is a very rare state which will likely never be triggered. If such a scenario ever occurs, while the first withdrawal execution is user-triggered - without a subsequent close - the next execution with now <math>\geq 8</math> ETH will result in <code>balance &lt; userCapital</code> (due to the absent of close call, <code>userCapital</code> remains un-zero'd).</p> <p>The result of this state is that the full 8 ETH will be distributed to the user side because <code>nodeShare</code> will remain unset in this specific control-flow.</p> <p>This issue has only been rated as a low issue due to the low likelihood.</p>
<b>Recommendations</b>	Consider further inspecting the likelihood of this issue, as the general idea of the <code>withdrawalBlock</code> implementation relies on such an assumption to happen. However, we did not find any real scenario where something like that would occur, unless someone donates $\geq 8$ ETH.
<b>Comments / Resolution</b>	Acknowledged.

<b>Issue_93</b>	Unexpected prohibition of bond reduction due to upgrade
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	Before Saturn, it is possible to reduce the bond amount for a node from 16 to 8. After Saturn, this is not possible anymore. It can happen that there are 16 ETH nodes which would like to decrease their bond at some point in the future but are now prohibited from doing so, after the Saturn upgrade.
<b>Recommendations</b>	Consider documenting this impact.
<b>Comments / Resolution</b>	Acknowledged. This is intentional - the severity has been downgraded to informational.

<b>Issue_94</b>	Permanently locked funds in case of self-mistake by Minipool owner for vacant minipool
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>Currently, there exists a specific sequence of actions for an owner to accidentally lock his own funds:</p> <ul style="list-style-type: none"> <li>- A vacant minipool is created and gets scrubbed before credentials changed</li> <li>- credential changed</li> <li>- owner calls close</li> <li>- minipool destroyed</li> <li>- validator only exits after close</li> </ul> <p>Using this sequence of actions, it can happen that funds remain locked in a minipool.</p>
<b>Recommendations</b>	We do not recommend a change, this requires active mistakes by the minipool owner, although not similar to sending funds to the dead address, as this might be unexpected.
<b>Comments / Resolution</b>	Partially resolved. Vacant minipool creation has been disabled, however, the chance exists that there have been vacant-not-activated minipoools before disabling the vacant minipool creation - hence partially resolved.

## RocketMinipoolFactory

The [RocketMinipoolFactory](#) contract is responsible for Minipool deployment via the [Clones](#) pattern. After the Saturn upgrade, no new Minipoools can be created anymore, hence, this contract will not be used.

**This contract has not been changed for the Saturn upgrade.**

### Privileged Functions

- `deployContract`

No issues found.

## RocketMinipoolManager

The **RocketMinipoolManager** stores an overview for overall, staking and finalized Minipoools. It exposes several view functions to fetch Minipoools per status and various other view functionalities related to Minipool settings.

**This contract has been changed for the Saturn upgrade.**

Core Invariants:

INV 1:

`createVacantMinipool/createMinipool/setMinipoolPubkey/updateNodeStakingMinipoolCount`  
must not be callable anymore

INV 2: `incrementNodeStakingMinipoolCount` must differentiate between 16 ETH and 8 ETH  
minipool bonds

INV 3: `incrementNodeStakingMinipoolCount` must increment `minipoools.staking.count`

INV 4: `_tryDistribute` must be triggered at the beginning of  
`increment/decrementNodeStakingCount`

Privileged Functions

- `createVacantMinipool`
- `createMinipool`
- `setMinipoolPubkey`
- `updateNodeStakingMinipoolCount`
- `incrementNodeStakingMinipoolCount`
- `decrementNodeStakingMinipoolCount`
- `incrementNodeFinalisedMinipoolCount`
- `removeVacantMinipool`
- `destroyMinipool`

Issue_95	Finalization DoS of 16 ETH minipools without active fee distributor via 1 wei donation
Severity	Low
Description	<p>Currently, if there is no active distributor (for NO with 16 ETH, old, minipools), it is possible to DoS closure of minipools via a 1 wei donation due to a balance check:</p> <pre> function _tryDistribute(address _nodeAddress) internal {     // Get contracts     RocketNodeDistributorFactoryInterface rocketNodeDistributorFactory =     RocketNodeDistributorFactoryInterface[getContractAddress("rocketNodeDistributorFactory")];     address distributorAddress =     rocketNodeDistributorFactory.getProxyAddress(_nodeAddress);     // If there are funds to distribute than call distribute     if [distributorAddress.balance &gt; 0] {         // Get contracts         RocketNodeManagerInterface rocketNodeManager =         RocketNodeManagerInterface[getContractAddress("rocketNodeManager")];         // Ensure distributor has been initialised          require[rocketNodeManager.getFeeDistributorInitialised(_nodeAddress), "Distributor not initialised"];         RocketNodeDistributorInterface distributor =         RocketNodeDistributorInterface[distributorAddress];         distributor.distribute();     } } </pre> <p>the <code>_tryDistribute</code> function is triggered on every minipool closure:</p> <pre> function decrementNodeStakingMinipoolCount(address _nodeAddress) override external onlyLatestContract["rocketMinipoolManager", address(this)] onlyRegisteredMinipool[msg.sender] { </pre>

	<pre>// Get contracts RocketMinipoolInterface minipool = RocketMinipoolInterface[msg.sender]; // Try to distribute current fees at previous average commission rate .tryDistribute[_nodeAddress];</pre> <p>And as highlighted in the first code snippet, the balance of this address is used as source of truth. In the case of a 1 wei donation while there is no deployed/init contract, it will simply revert and force the NO to create a distributor.</p>
<b>Recommendations</b>	This can be simply bypassed by creating such a distributor and since it is even favored that everyone has a distributor, we do not recommend a change. However, for users that are not aware of this, it can become a temporary issue.
<b>Comments / Resolution</b>	Acknowledged. There are very few node operators without an active fee distributor and in this scenario they just need to initialise their fee distributor before finalisation

<b>Issue_96</b>	Duplicate deletion of <code>validator.minipool</code> in dissolve -> <code>destroyMinipool</code> control-flow
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	The <code>validator.minipool</code> state is deleted twice during the dissolve and subsequent <code>destroyMinipool</code> flow.  This is a redundant operation.
<b>Recommendations</b>	Consider acknowledging this issue as it does not expose any harm.
<b>Comments / Resolution</b>	Acknowledged.

<b>Issue_97</b>	Potentially incorrect version display
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	Within the resolution round, the version was changed from 5 to 6. However, most likely, version 5 has never been deployed which makes the change from version 5 to version 6 redundant, as version 6 was just a fix to version 5 which means it is theoretically still version 5.
<b>Recommendations</b>	Consider acknowledging this issue.
<b>Comments / Resolution</b>	Acknowledged.



## RocketMinipoolPenalty

The [RocketMinipoolPenalty](#) contract is a simple intermediate contract which stores the penalty rate for a minipool and is triggered via the [NetworkPenalties](#) contract [trusted node voting]

**This contract has not been changed for the Saturn upgrade.**

Core Invariants:

INV 1: Only the guardian can set the maximum penalty

INV 2: setPenalty must only be triggered upon quorum reach for trusted node count

Privileged Functions

- setMaxPenaltyRate
- setPenaltyRate

No issues found.

## RocketMinipoolQueue

The [RocketMinipoolQueue](#) contract is handling the queue mechanism for Minipools, including backwards compatibility with legacy-kind minipools. Within the current state of RocketPool, it is solely used to organize the queue for variable type minipools.

**This contract has not been changed for the Saturn upgrade.**

**Please note that the current audit is only conducted around variable type minipools.**

**Core Invariants:**

INV 1: Each loop within `dequeueMinipools` must dequeue the current head of the queue

INV 2: Only `MinipoolDeposit.Variable` type minipools must be in the queue

INV 3: [`RocketMinipoolQueue.getEffectiveCapacity`](#) must only include `variableCapacity` due to legacy pools not being in queue anymore

**Privileged Functions**

- `enqueueMinipool`
- `dequeueMinipoolByDepositLegacy`
- `dequeueMinipools`
- `removeMinipool`

No issues found.

## RocketMinipoolStorageLayout

The [RocketMinipoolStorageLayout](#) contract represents the storage layout which is inherited by the [MinipoolDelegateBase](#) contract. It includes current storage variables as well as storage variables which are unused and only present for backwards compatibility.

**This contract has not been changed for the Saturn upgrade.**

### Privileged Functions

- none

No issues found.

# Network

## RocketNetworkBalances

The [RocketNetworkBalances](#) contract enables trusted nodes to submit state balances for a specific block. The following states can be set:

- **network.balances.updated.block**: The last updated block. It is used to make sure future submissions do not apply for past blocks
- **network.balances.updated.timestamp**: The last updated timestamp. It is used to determine the [minimumTimestamp](#) which must have passed for the next update to be allowed.
- **network.balance.total**: The total network ETH balance. This data is fetched from a script but is meant to include only user-related ETH (excluding node bonds). It is used for determining shortfall within the [TokenRETH](#) contract and the general exchange rate between ETH and RETH
- **network.balance.staking**: [network.balance.total](#) excluding ETH in: [DepositPool](#), [TokenRETH](#), [Minipools](#), [Megapools](#). It is used to determine the ETH utilization rate which is purely from informational nature.
- **network.balance.reth.supply**: The total circulating RETH supply. It is used to determine the exchange rate between ETH and RETH

This contract has been changed for the Saturn upgrade.

### Appendix: Balance Update Process

Any trusted node can call the [submitBalances](#) function with the corresponding parameters and once the [nodeConsensusThreshold](#) is met, it will automatically execute the balance update and change the state variables for the said. A safeguard has been implemented which ensures that the exchange rate between ETH and RETH cannot be changed more than [getMaxRethDelta](#).

## Core Invariants:

INV 1: During updateBalances, the exchange rate between ETH and RETH cannot change more than getMaxRethDelta [RPIP-61]

INV 2: Balances cannot be submitted for a future block

INV 3: Balances cannot be submitted for a block which is smaller than lastBalancesBlock

INV 4: Same address can only submit once for specific setup

INV 5: updateBalances is only allowed if 95% of the frequency has passed since the last updateBalances call [RPIP-61]

## Privileged Functions

- submitBalances

<b>Issue_98</b>	Division by zero error if currentTotalETHBalance > 0 and RETH supply = 0
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>Currently, it is possible that such a state as described in the title happens and thus the operation reverts.</p> <p>This issue is only rated as informational severity because such a state is very rare to happen. This could for example happen if the protocol is completely emptied, followed by a 1-wei donation to the <b>TokenRETH</b> contract, resulting in ETH balance = 1 wei and RETH supply = 0</p>
<b>Recommendations</b>	Consider acknowledging this issue.
<b>Comments / Resolution</b>	Acknowledged.

<b>Issue_99</b>	Missing invariant enforcement that for <code>_totalETH</code> being larger than <code>_stakingETH</code>
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	Currently, there is no explicit enforcement that <code>_totalETH &gt;= _stakingETH</code> .  It is however expected that trusted nodes properly validate their data.
<b>Recommendations</b>	Consider acknowledging this issue.
<b>Comments / Resolution</b>	Resolved.

## RocketNetworkFees

The `RocketNetworkFees` contract exposes logic to calculate the fee for a node based on user/node demand. The `getNodeFee` function is considered during the initialization of a new Minipool or during bond reduction.

Since both these control-flows are disabled, this contract remains essentially unused.

**This contract has not been changed for the Saturn upgrade.**

### Appendix: Fee by demand

The fee-by-demand algorithm dynamically adjusts the node commission rate based on the network's current ETH demand:

#### Demand definition:

- `nodeDemand = depositPoolBalance - minipoolCapacity`
- Positive demand means *too much user ETH waiting for nodes* [high demand for node operators].
- Negative demand means *too much available node capacity* [low demand for node operators].

#### Fee interpolation logic:

- Demand is normalized relative to the configured `demandRange`
- If normalized demand is zero → return `targetFee`.
- If `demand ≥ range` → clamp to `maxFee` (positive) or `minFee` (negative).
- Otherwise, interpolate smoothly using a cubic curve ( $t^3$ ) between the target and the extremes, giving non-linear sensitivity near equilibrium.

#### Summarized:

- a] Positive demand increases the node fee toward `maxFee`.
- b] Negative demand decreases it toward `minFee`.
- c] The transition curve is cubic, making small demand changes near balance have only a mild fee impact, while large imbalances move fees aggressively.



## Core Invariants:

INV 1: High node demand must result in lower node fees

INV 2: Low node demand must result in higher node fees

INV 3: If `nodeDemand = 0`, must return `targetFee`

INV 4: The fee curve must be non-linear and steeper towards the extremes

## Privileged Functions

- none

No issues found

## RocketNetworkPenalties

The [RocketNetworkPenalties](#) contract allows trusted nodes to submit penalty requests for Minipools. It follows the same pattern as most other submit/vote related contracts and allows each trusted node to submit only once for a specific set of Minipool/block, preventing double submissions.

A submission becomes valid if the [nodePenaltyThreshold](#) has been reached and for the first 2 counts, no impact is registered. Starting from the third count for a Minipool, a penalty of  $(\text{count} - 2) * \text{perPenaltyRate}$  is applied.

The contract follows the exact same maximum clamping approach as the [MinipoolPenalties](#) contract and we refer to the specific appendix.

**This contract has been changed for the Saturn upgrade.**

**Core Invariants:**

INV 1: penalise cannot be called twice with the same minipool/block parameters by the same node

INV 2: During applyPenalty, currentMaxPenalty must be larger than zero

INV 3: Within penaltyMaximumPeriod, it is not allowed to aggregate penalty more than maximumPenaltyCount

INV 4: The first two penalties must not have any effect on a Minipool's penalty rate

**Privileged Functions**

- submitPenalty

\*Please see [MegapoolPenalties](#) contract for shared issues.

No issues found.

## RocketNetworkPrices

The `RocketNetworkPrices` contract allows trusted nodes to submit price requests for RPL. It follows the same pattern as most other submit/vote related contracts and allows each trusted node to submit only once for a specific set of `_block`, `_slotTimestamp`, `_rplPrice` preventing double submissions.

A submission becomes valid if the `nodePenaltyThreshold` has been reached and for the first 2 counts, no impact is registered. Starting from the third count for a Minipool, a penalty of  $(\text{count} - 2) * \text{perPenaltyRate}$  is applied.

This contract has not been changed for the Saturn upgrade.

Core Invariants:

INV 1: Price cannot be submitted for a future block

INV 2: Price cannot be submitted for a block which is smaller than lastBalancesBlock

INV 3: Same address can only submit once for specific setup

### Privileged Functions

- `submitPrices`

<b>Issue_100</b>	Lack of <code>SettingsNetwork.getSubmitPricesFrequency</code> incorporation into <code>submitPrice</code> function
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	Currently, the <code>submitPrice</code> function does not incorporate the <code>getSubmitPricesFrequency</code> value, allowing to submit prices in any frequency.
<b>Recommendations</b>	We recommend assessing whether this is a design choice. We don't see any disadvantage with that methodology as it currently is.
<b>Comments / Resolution</b>	Acknowledged. This is intentional.

## RocketNetworkRevenues

The [RocketNetworkRevenues](#) contract handles setting and getting of the share for:

- nodeShare
- voterShare
- protocolDAOShare

### Appendix: Split calculation

For this section, we refer to the capital dispersal appendix within the [MegapoolDelegate](#) section.

### Appendix: Average Share Calculation

Throughout the lifecycle shares can be adjusted via governance votes. For example, it may be possible to have nodeShare = 0.05 ETH [5%] at the beginning and then later changed to 0.1 ETH [10%].

Within the Megapool contract, rewards are always calculated based on the shares since the [lastDistributionBlock](#). That means, if the nodeShare has changed between [lastDistributionBlock; block.timestamp], the average needs to be calculated to ensure a fair distribution.

The logic is as follows:

- a) Fetch the snapshot at the latest checkpoint
- b) Fetch the share at the latest checkpoint
- c) Calculate the duration from the latest checkpoint to now
  - i)  $\text{blockDuration} = \text{block.number} - \text{checkpointBlock}$
- d) Calculate the accumulator from the latest checkpoint until now
  - i)  $\text{currentAccum} = \text{checkpointValue} + [\text{share} * \text{blockDuration}]$
- e) Fetch the nearest checkpoint before or at sinceBlock
- f) Fetch the share at this checkpoint
- g) Calculate the duration from this checkpoint to the latest checkpoint
  - i)  $\text{blockDuration} = \text{sinceBlock} - \text{checkpointBlock}$
- h) Calculate accumulator until sinceBlock
  - i)  $\text{pastAccum} = \text{checkpointValue} + [\text{sharePast} * \text{blockDuration}]$
- i) Calculate duration from now to sinceBlock
  - i)  $\text{duration} = \text{block.number} - \text{sinceBlock}$

- j] Calculate average
  - i]  $\text{average} = [\text{currentAccum} - \text{pastAccum}] / \text{duration}$

Summarized, it will simply calculate the accumulator until `block.timestamp` and the accumulator until `sinceBlock`, then use the difference and divide it by the blocks passed between `block.number` and `sinceBlock` to then get the average share between `block.number` and `sinceBlock`.

#### Core Invariants:

INV 1: The average must always be from `sinceBlock` to `block.number`

INV 2: `_setShare` must always calculate the accumulator until `block.number`

INV 3: `valueKey` is determined as `bytes32(uint256[shareKey] + block.number)`

INV 4: Each pushed checkpoint must have a corresponding share at `valueKey`

INV 5: Whenever a new share is stored, the accumulator up to the current `block.number` for the share is pushed

#### Privileged Functions

- `setNodeShare`
- `setVoterShare`
- `setProtocolDAOShare`

Issue_101	Varying block times may produce unfair share result
Severity	Low
Description	<p>Currently, there is a potential vulnerability if block times vary or are changed in a future upgrade.</p> <p>For example, if the block time is increased, the same time period will produce less blocks and therefore the impact at one specific share checkpoint will be less than expected.</p> <p>This will become especially problematic in the scenario where a mainnet upgrade changes the block duration.</p>
Recommendations	<p>To fix this issue, it is required to use block.timestamp instead of block.number. However, this is not practical because it requires a change in multiple contracts [see below explanation]. And if these are switched from block.number to block.timestamp, it will inherently break the state post-upgrade.</p> <p><b>Explanation of required changes:</b></p> <p>This will need a change in multiple contracts and additional testing resources to make sure all cross-contract interactions handle this change correctly.</p> <p>Additional resources that go beyond resolution resources will be required because this will go deep into the NetworkSnapshots contract which is used by:</p> <ul style="list-style-type: none"> <li>- MinipoolManager</li> <li>- DepositPool</li> <li>- MegapoolPenalties</li> <li>- NetworkPrices</li> <li>- NetworkRevenues</li> <li>- NetworkVoting</li> <li>- NodeManager</li> <li>- NodeStaking</li> </ul> <p>Due to the fact that all these contracts will need to be adjusted if</p>

	<p>the switch is made from block.number to block.timestamp within the NetworkRevenues contract, we recommend acknowledging this issue. Furthermore, the impact is small.</p> <p>It is furthermore, as already mentioned, not possible as historical data is stored with block.number and now applying block.timestamp will break fetching mechanisms.</p>
<b>Comments / Resolution</b>	Open. A time-based version of the snapshot mechanism has been introduced. This needs to be fully reviewed to make sure no edge-cases are introduced. There are no immediate obvious issues with this mechanism though. Furthermore, capital ratio mechanism has been incorporated which needs to be fully reviewed.
<b>Resolution 2 / Follow-Up</b>	Resolved, the contract now uses block.timestamp instead of block.number and this will not expose any backwards issues because this contract has not been deployed pre-Saturn.

<b>Issue_102</b>	Incorrect comment mentions block instead of timestamp [FOLLOWUP]
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	Throughout the contract, multiple comments still refer to block instead of timestamp:  <i>/// @notice Returns the average capital ratio of the given node operator since a given block</i>
<b>Recommendations</b>	Consider referring to timestamp instead of block.
<b>Comments / Resolution</b>	Resolved.

## RocketNetworkSnapshots

The [RocketNetworkSnapshots](#) contract serves as a registry for snapshots related to blocks for different specific values. It is used for:

- ETH borrowed for Megapools
- Penalty for Megapools (running total)
- ETH borrowed for Minipools
- activeMinipoolCount
- Penalty for Minipools (running total)
- RPL price
- Shares for Megapool reward calculation
- Voting delegate
- Staked amount for node

This contract has been changed for the Saturn upgrade.

### Appendix: Binary search mechanism

Values are appended as checkpoints {block, value}. If the same block is written twice, the last write overwrites that block's value.

To answer “value at block B”, the contract runs a binary search to find the first checkpoint with block > B [the insertion point], then returns the previous checkpoint [the last block  $\leq B$ ]. This is a deviation from the standard binary search mechanism.

[lookupRecent](#) is identical but starts its binary search in the recent tail of the list (based on a recency count of checkpoints) to speed up near-now queries; if the target isn't in that tail, it automatically searches the older half.

[lookupCheckpoint](#) was a simple addition which returns the full Checkpoint info instead of only the value.

### Core Invariants:

INV 1: checkpoint submissions during the same block must update instead of push



## Privileged Functions

- push

No issues found.

## RocketNetworkVoting

The [RocketNetworkVoting](#) contract is responsible for fetching the voting power of a node at a specific block and allows for changing the delegate address.

**This contract has been changed for the Saturn upgrade.**

### Appendix: Voting Power Calculation

The voting power at `_block` is computed from snapshots in three stages:

#### a) Pull historical inputs (at `_block`):

- Borrowed ETH for Minipools
- Active minipools
- Megapool ETH bonded
- RPL price (ETH per RPL)
- Node's RPL stake
- Max stake percent

#### b) Compute the node's total bonded ETH:

```
> totalETHStaked = activeMinipools * 32 ether  
  
> bondedETH = totalETHStaked - borrowedETH + megapoolETHBonded
```

#### c) Cap how much RPL can count

```
> maximumStake = bondedETH * maxStakePercent / rplPrice  
  
> effectiveStake = min[rplStake, maximumStake]
```

#### d) Return voting power:

```
> votingPower = sqrt[effectiveStake * 1e18] [square-root based weight]
```

## Core Invariants:

INV 1: VP must be clamped with maximumStake

## Privileged Functions

- none

<b>Issue_103</b>	Incorrect NATSPEC for activeMinipool purposes
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>The following comment is incorrect:</p> <pre>// Get active minipools to calculate borrowed ETH key = keccak256(abi.encodePacked("minipools.active.count", _nodeAddress));</pre> <p><i>active minipools are used to calculate bonded ETH:</i></p> <pre>uint256 totalETHStaked = [activeMinipools * 32 ether];</pre>
<b>Recommendations</b>	Consider changing this comment.
<b>Comments / Resolution</b>	Resolved.

## Node

### RocketNodeDeposit

The `RocketNodeDeposit` contract is the entry contract for nodes to create validators for a Minipool.

This can be done with three different functions:

- a) `deposit`: This is the standard function which requires the `msg.value` to match exactly the desired bond, which is by default 4 ETH
- b) `depositWithCredit`: This function is similar to the standard deposit function with the difference that `msg.value` can be below the bond amount. It will then simply take any remaining credit or balance to balance out the deficit
- c) `depositMulti`: This function allows for multiple deposits at the same time while also including the crediting logic

Furthermore, it is possible to deposit ETH which can then be used as balance for validator creation or can be withdrawn again.

**This contract has been changed for the Saturn upgrade.**

**Core Invariants:**

INV 1: `node.deposit.credit.balance` must only be increased upon vacant Minipool promotion and Megapool bond reduction or dequeue

INV 2: `RocketNodeDeposit.depositETHFor` must be considered as `balanceToUse`

INV 3: `RocketNodeDeposit.deposit`; if `bondAmount < msg.value`, the leftover must be taken from credit or from eth balance

INV 4: `useCreditOrBalanceIfRequired` must return the amount which is withdrawn from `RocketVault`

INV 5: Each node can only have one Megapool

## Privileged Functions

- increaseDepositCreditBalance

<b>Issue_104</b>	No bond requirement in pre-upgrade codebase allows for spamming vacant minipools with the goal to pass without scrubbing
<b>Severity</b>	<b>Medium</b>
<b>Description</b>	<p>Before the upgrade, it is possible to spam the creation of vacant minipools because no bond / lock /... is required.</p> <p>An attacker can spam hundreds or thousands of minipools with the target to flood scrubbing abilities, just to pass the scrubbing time for one or more minipools and then create a vacant minipool without actual backing.</p> <p>It has to be noted that vacant minipools cannot be created anymore after saturn which makes this attack not actively executable anymore. However, it can theoretically still happen that this state occurs [an attacker will just create hundred vacant minipools and then the update happens], which means this issue needs to be raised in this report as well.</p> <p><b>This issue is live in the current deployed version.</b></p>
<b>Recommendations</b>	Consider disabling vacant minipools.
<b>Comments / Resolution</b>	Resolved, Bailsec has reported another live-issue during the audit and Rocketpool disabled the creation of vacant minipools immediately.

<b>Issue_105</b>	Missing msg.value check within <code>depositMulti</code> can result in ETH donation
<b>Severity</b>	<b>Medium</b>
<b>Description</b>	Currently, it is possible to provide a higher msg.value than what is actually required. No refund is issued and this will be simply donated. The caller will lose his funds.
<b>Recommendations</b>	Consider implementing a strict check which ensures that the msg.value matches with what is required and revert if that is not true.
<b>Comments / Resolution</b>	Resolved, a check has been implemented which ensures msg.value cannot be larger than the required bond amount.

<b>Issue_106</b>	Incorrect error wording within <code>increaseDepositCreditBalance</code>
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>The access control mechanism within <code>increaseDepositCreditBalance</code> has been tightened to only allow registered minipools calling it.</p> <p>However, the revert error still indicates potentially outdated network contract, which refers to the previous check of:</p> <pre>getBool(keccak256(abi.encodePacked("contract.exists", msg.sender)))</pre>
<b>Recommendations</b>	Consider re-wording the error message to reflect that now only registered minipools can call this function.
<b>Comments / Resolution</b>	Acknowledged.

## RocketNodeDistributor

The [RocketNodeDistributor](#) contract is considered as the fee recipient / coinbase address for the NO and its related validators. It is expected that any fees are relayed to this contract and in case of no-compliance, slashing occurs. As an alternative of using this contract, NOs can also relay their fees to the [SmoothingPool](#) and then receive rewards via the [MerkleDistributor](#) which relies on an off-chain calculation mechanism.

This contract is a simple proxy contract which is meant to point to the [RocketNodeDistributorDelegate](#) contract, which forms the implementation and exposes the logic. The implementation is stored in the global [RocketStorage](#) contract under “[rocketNodeDistributorDelegate](#)”. Therefore, governance can change the implementation contract globally, at any time.

**This contract has not been changed for the Saturn upgrade.**

### Privileged Functions

- none

No issues found.



## RocketNodeDistributorDelegate

The `RocketNodeDistributorDelegate` contract is meant to be the implementation for the `RocketNodeDistributor` proxy contract.

It is responsible for fairly distributing the user and node share based on the overall native ETH in the contract.

The native ETH is received via fees and MEV rewards from validators.

**This contract has been changed for the Saturn upgrade.**

### Appendix: Distribution Calculation

The contract exposes a `getNodeShare` function which calculates the `nodeShare` based on the native ETH balance. This is done via two ways:

- a) The `collateralRatio` [bond <-> user capital] for a NO is fetched and used to determine the `nodeBalance` and `userBalance`.
- b) On top of the `nodeBalance`, a fee in the form of the `averageNodeFee` is applied on the `userBalance` and added towards the `nodeBalance`. This can be considered as a commission for the NO.

### Core Invariants:

INV 1: In case where a user calls `distribute`, `nodeShare` must be added to unclaimed rewards instead of a direct transfer execution.

### Privileged Functions

- none

No issues found.

## RocketNodeDistributorFactory

The [RocketNodeDistributorFactory](#) contract is a simple factory contract which deploys RocketNodeDistributor instances for a NO.

This contract has not been changed for the Saturn upgrade.

### Privileged Functions

- `createProxy`

No issues found.

## RocketNodeDistributorStorageLayout

The [RocketNodeDistributorStorageLayout](#) contract is inherited by the [RocketNodeDistributorDelegate](#) contract and exposes a minimalistic storage layout.

This contract has been changed for the Saturn upgrade.

### Privileged Functions

- none

No issues found.

## RocketNodeManager

The **RocketNodeManager** contract handles the registration and settings for nodes, such as:

- Distributor
- Timezone
- RPL withdrawal address
- Reward network
- SmoothingPool state
- Express tickets
- Unclaimed rewards

This contract has been changed for the Saturn upgrade.

### Appendix: Average Node Fee

A) Fetch the node's staking minipool counts for each bond size [16 ETH, 8 ETH] and the network launchAmount [32 ETH]. If the node has no staking minipoools, return 0.

B) Build weights per bond size: for each size, compute:

- > `weightRaw[i] = [launchAmount - depositSize[i]] * depositCounts[i]`
- total user-side ETH paired with the node at that size.

Sum all weightRaw to get weightTotal, then normalize:

- > `weight[i] = weightRaw[i] * 1e18 / weightTotal`

C) For each size with `depositCounts[i] > 0`, read the stored fee numerator:

- 16 ETH: `key node.average.fee.numerator`
- 8 ETH: `key node.average.fee.numerator,[size]`

Each numerator represents the sum of node fees (1e18-scaled) across the node's staking minipools of that size. It is important to mention that this must always align with the `depositCount`.

D) Compute the per-size average fee by dividing by the count, then apply the weight and accumulate:

- > `weightedAverage += [numerator * weight[i]] / depositCounts[i]`

E) Return the final average node fee (1e18-scaled) by de-scaling the accumulator:



Net effect: it's a user-capital-weighted average of the node's commission across 16-ETH and 8-ETH minipools, where each size's fee is first averaged over the node's active minipools of that size, then blended using the user ETH paired at that size as the weight.

#### Core Invariants:

INV 1: After registration, a node has no tied node.rpl.withdrawal.address

INV 2: Express tickets can only be provisioned once for nodes that have registered pre saturn upgrade

INV 3: registerNode must increase node.count

INV 4: registerNode must delegate to self

INV 5: reward network for a node can only set to an enabled network

INV 6: setSmoothingPoolRegistrationState can only be called if [rewardInterval since lastChange has passed](#)

INV 7: node.smoothing.pool.state must be false by default

INV 8: Everyone can register as a node

#### Privileged Functions

- useExpressTicket
- provisionExpressTicket
- refundExpressTicket

Issue_107	initialiseFeeDistributor function can be abused to inflate average fee
Severity	High
Description	<p>The <code>initialiseFeeDistributor</code> function is solely callable for old nodes with 16 ETH minipools that have not yet initialized their fee distributor.</p> <p>For any other nodes, this function is not callable because it was inherently triggered during the control-flow of the minipool creation pre Saturn. Now after Saturn, minipools cannot be created anymore.</p> <p>A mistake within the determination process of active minipools allows a malicious NO to inflate their fees:</p> <pre><code>uint256 count = rocketMinipoolManager.getNodeMinipoolCount[msg.sender];</code></pre> <p>The mistake lies within this line, as this includes all minipools, including those which are not staked. And later, there is no sanity check which excludes minipools.</p> <p>If we now examine the flow when a minipool is finalized, we will realize that <code>decrementNodeStakingMinipoolCount</code> decreases <code>minipools.staking.count</code>.</p> <p>If a malicious NO has multiple 16 ETH minipools and finalizes them, followed by a call to <code>initialiseFeeDistributor</code>, it will now set <code>node.average.fee.distributor</code> which includes the finalized minipools.</p> <p>Now further examining the <code>getAverageNodeFee</code> function, we see that the divisor is <code>getNodeStakingMinipoolCountBySize</code>, which *excludes* finalized minipools.</p> <p>The result will be that the divisor is smaller than expected and thus inflate the overall result of the <code>getNodeAverageFee</code> function.</p> <p>This exploit essentially allows the NO to now inflate their share for</p>

	<p>distributions via the distributor contract.</p> <p><b>This issue is live in the current deployed version.</b></p>
<b>Recommendations</b>	<p>Consider making the validation within initialiseFeeDistributor more strict and explicitly exclude any finalized minipools:</p> <p><code>Minipool.getFinalised</code></p>
<b>Comments / Resolution</b>	<p>Acknowledged, the Rocketpool team added the following comment:</p> <p>"We have investigated and confirmed this has not been exploited on mainnet and there are only a few remaining NOs who have yet to initialise their fee distributor that may still be affected."</p>

<b>Issue_108</b>	Lack of validation for <code>addUnclaimedRewards</code>
<b>Severity</b>	Low
<b>Description</b>	Currently, it is possible to call the <code>addUnclaimedRewards</code> function with an address that is not registered as node. These funds will be lost until the address registers as a node and in case the address is a contract, funds may be lost.
<b>Recommendations</b>	Consider validating the address.
<b>Comments / Resolution</b>	Resolved

<b>Issue_109</b>	User can claim unlimited express tickets by using <code>registerNode</code> with different addresses
<b>Severity</b>	<b>Low</b>
<b>Description</b>	<p>Whenever a new node is registered, two express tickets will be assigned. This opens up a simple sybil-attack vector, as now a user can use different addresses and grab free express-tickets.</p> <p>It has to be mentioned that this means a user will be required to create more than 1 megapool, as these tickets are tied to the address.</p>
<b>Recommendations</b>	Consider keeping this in mind. There is no trivial fix.
<b>Comments / Resolution</b>	Acknowledged. The Rocketpool team mentioned that there is disincentive for doing this as each separate node has a separate Megapool and thus would require multiple distribute transactions to distribute rewards.

<b>Issue_110</b>	Strict validation for timezone selection
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>Currently, the validation for timezones is as follows:</p> <pre>require[bytes[_timezoneLocation].length &gt;= 4, "The timezone location is invalid"];</pre> <p>This means that timezones such as UTC, GMC, .. cannot be registered.</p>
<b>Recommendations</b>	Consider loosening this check.
<b>Comments / Resolution</b>	Acknowledged.

<b>Issue_111</b>	UX for first-time express usage is suboptimal
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	Currently, the provision of express ticket is only allowed for the DepositPool contract. This means any address without a ticket has to deliberately assign a validator (either express or standard) and only then the ticket gets claimed.
<b>Recommendations</b>	Since this is a slight downside of the UX, we do not recommend a change such as marking this permissionless, as the permissioned nature is in favor of defensive coding.
<b>Comments / Resolution</b>	<p>Resolved. The Rocketpool team added the following comment:</p> <p>"We have introduced a method to allow NOs to manually provision their express tickets. This was due to the fact that provisioning was happening on first deposit after Saturn, however the number of tickets received is based on number of Minipools. A NO will want to exit Minipools and then deposit for Megapools. So the order of provisioning would result in them not receiving the correct number of tickets."</p> <p>We however don't think this reasoning is fully accurate since users could likely just keep their minipools, create a megapool and receive tickets and then exit their minipools. This way would still allow users to receive their correct ticket amount.</p> <p>It has to be mentioned that the permissionless nature of this function now potentially introduces side-effects which are not considered.</p>

Issue_112	Factory upgrade will prevent <code>addUnclaimedRewards</code> call
Severity	<b>Informational</b>
Description	<p>The <code>addUnclaimedRewards</code> function access control has been tightened to only allow distributors to call it. This is achieved by ensuring that only the corresponding proxy contract can execute the call.</p> <p>In the scenario where the distributor factory is upgraded, this will result in incompatibility for old distributors.</p>
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

## RocketNodeStaking

The [RocketNodeStaking](#) contract enables registered nodes to stake/unstake/withdraw/lock/unlock RPL tokens. Locking and unlocking solely happens if a node interacts with the Protocol DAO by submitting a proposal or challenging a proposal - likewise for burn - where a part of the locked funds are then burned.

It furthermore allows any node to allow other addresses to stake for self or to lock RPL (which is used during protocol proposals or proposal challenges).

**This contract has been changed for the Saturn upgrade.**

### Appendix: Migration

The contract implements a bucket which is related to pre-saturn staked RPL:  
[rpl.legacy.staked.node.amount](#).

This bucket is filled via the [migrateLegacy](#) function before any new state transition for a user's RPL is happening. It makes a distinct difference between old and new staked RPL and will inevitably be accounted for without any bypass method.

The [rpl.legacy.staked.node.amount](#) is used for potential minipool backing (which should be disabled in this commit) and it is not possible for users to increase [rpl.legacy.staked.node.amount](#) after the saturn upgrade.

### Appendix: Lock

The contract allows for locking RPL. This functionality is solely used by the Protocol DAO whenever a new proposal is created or whenever a proposal is challenged. The required [proposalBond](#) or [challengeBond](#) is then simply locked for a user.

### Appendix: Stake Process

Users have the ability to stake RPL which will count towards [totalKey](#) and [totalMegapoolKey](#) and will be stored within the [RocketNetworkSnapshots](#) contract. It is important to mention - while the contract makes a difference between megapool and minipool RPL (legacy), the key which is used for storing into the [RocketNetworkSnapshots](#) contract must always include both stakes and must properly account for any stake as well as unstake process

## Appendix: Withdrawal Process

The withdrawal process is a two-step process which requires first an unstake execution that decreases the RPL stake and pushes the amount into the `rpl.megapool.unstaking.amount` bucket.

It can then be fully withdrawn once the `unstakingPeriod` has passed. Each unstake resets the `lastUnstakeTime` for a node and requires to wait another full `unstakingPeriod` - even if a previous unstake already partially went through the said period.

## Appendix: Slashing

The `slashRPL` function is solely called by the Minipool, whenever a NO is being slashed. It will simply decrease a node's legacy RPL stake and transfer RPL towards the `RocketAuctionManager` contract.

## Core Invariants:

INV 1: slashing must only be applied on the legacy RPL amount

INV 2: `rpl.staked.node.amount` must always represent the megapool and minipool (legacy) RPL stake

INV 3: `increaseNodeRPLStake` must increase `totalKey` and `totalMegapoolKey`

INV 4: `decreaseNodeRPLStake` must favor legacy stake decrease

INV 5: `getNodeLegacyStakedRPL` must return always the staked amount from previous implementation

INV 6: [can never decrease more RPL](#) than what is staked by a user

INV 7: [migrateLegacy](#) must be called before all mutating functions

INV 8: `transferRPL/burnRPL` must only be called by the Protocol DAO module

INV 9: `decreaseNodeMegapoolRPLStake` [must not decrease legacyStakedRPL](#)

## Privileged Functions

- `lockRPL`
- `unlockRPL`
- `transferRPL`

- burnRPL
- slashRPL

Issue_113	Ratio shifting from 8 to 16 ETH bond can be used to manipulate the return value of <code>getNodeETHCollateralisationRatio</code>
Severity	High
Description	<p>The <code>getNodeETHCollateralisationRatio</code> function returns the ratio of bond to borrow which is then used within <code>NodeDistributorDelegate.getNodeShare</code>. This determines how much fees will go towards the NO and towards users.</p> <p>There are currently two minipool types:</p> <ul style="list-style-type: none"> <li>- 16 ETH bond</li> <li>- 8 ETH bond</li> </ul> <p>For example, with a 8 ETH bond, it will have a user share of 24 ETH which means more fees go to users than to the NO. There is however an edge-case which allows to manipulate this ratio at no cost, allowing users to shift their ratio towards 16 ETH ratio, even if they only own 8 ETH minipools.</p> <p>This issue relies on the fact that anyone can create a vacant minipool at no cost with <code>bondAmount = 16 ETH</code>. The creation of this vacant minipool will then immediately flow into the ETH bonded/borrowed accounting and even if the minipool is dissolved because it does not represent a real validator on the beacon chain [and gets scrubbed], only the close call will reset the impact of the minipool on ETH bonded/borrowed accounting. However, the close function is permissioned to the owner which means the owner can just decide not to close such a minipool.</p> <p>It is therefore possible to create multiple vacant 16 ETH minipools while owning only 8 ETH minipools and that way, stealing a part of the fees which are meant to be entitled to the users.</p> <p><b>This issue is live in the current deployed version.</b></p>

<b>Recommendations</b>	<p>The solution for this issue is to simply make the close call permissionless [evtl. side effects need to be considered].</p> <p>However, due to the way how the upgrade mechanism works, this will not help with the current live deployment version.</p> <p>We therefore recommend analyzing the situation, penalize any actor that has exploited this vulnerability and also prevent the creation of vacant minipools.</p>
<b>Comments / Resolution</b>	Resolved, the team has followed the recommended steps.

<b>Issue_114</b>	Revert of <code>stakeRPL/unstakeRPL/withdrawRPL/unstakeLegacyRPL</code> if <code>RPLWithdrawalAddress</code> is set
<b>Severity</b>	Low
<b>Description</b>	Currently, the <code>onlyRegisteredNode</code> modifier within the mentioned functions prevents the direct call of these, as it will require that the caller is the <code>rplWithdrawalAddress</code> , if the said address is set.
<b>Recommendations</b>	Consider using the “for” functions in that case.
<b>Comments / Resolution</b>	<p>Acknowledged, the Rocketpool considers this as non-issue and added the following comment:</p> <p>“This is intentional. These functions are intended to be called by the node address. If <code>RPLWithdrawalAddress</code> is set, these functions can no longer be called by the node directly”</p> <p>We however believe this issue is still valuable to mention because it may result in unexpected reverts for users. We recommend adding proper documentation for this scenario.</p>

Issue_115	Cooldown bypass via <code>transferRPL</code>
Severity	Low
Description	<p>The <code>transferRPL</code> function decreases RPL from one address and allocates it to another address. It is used in different mechanics within the DAO component. Whenever this happens, indeed a share is burned and not all the RPL is re-allocated.</p> <p>However, it is still theoretically possible to bypass the cooldown period via such transfers - even though that means a loss is encountered.</p>
Recommendations	We do not recommend a change as most likely the fee which is applied does not make such an attack realistic for anyone.
Comments / Resolution	Acknowledged.

<b>Issue_116</b>	Permanent reset of <code>lastUnstakeTime</code> can be confusing
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	Whenever an unstake happens, the <code>lastUnstakeTime</code> setting is reset to the current block.timestamp. This can result in side-effects for users that do not expect such a behavior.
<b>Recommendations</b>	Consider carefully documenting this behavior on the frontend.
<b>Comments / Resolution</b>	<p>Acknowledged, the Rocketpool considers this as non-issue and added the following comment:</p> <p>“`lastUnstakeTime` keeps track of the time a node operator last unstaked. Setting it to the current time when unstaking is the expected behaviour of this variable. I don't understand why this behaviour would be unexpected.”</p> <p>In fact, we are fully aligned with this comment. However, the scenario we attempt to describe is that a user may withdraw only a small amount which fully resets the timer - requiring to wait for another period. This is why we do not consider it as an issue in itself but a potential footgun for users that are not aware of this behavior. A simple documentation of this behavior should provide clarity</p>

<b>Issue_117</b>	Lack of update for <code>rpl.staked.node.time</code>
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	Currently, the <code>rpl.staked.node.time</code> value is not updated. Therefore, it will point to the last staked time before the Saturn upgrade.
<b>Recommendations</b>	Consider acknowledging this issue if this value is not actively used anywhere.
<b>Comments / Resolution</b>	Resolved. a new [old] cooldown mechanism has been re-implemented.

Issue_118	Mixup of legacy and megapool RPL in case of transfer
Severity	Informational
Description	<p>The <code>transferRPL</code> function is decreasing the legacy RPL amount and then the megapool RPL amount, but generally it favors first the decrease of legacy RPL.</p> <p>However, when granted to the recipient it only increases the megapool RPL amount and not the legacy RPL amount.</p> <p>This can be used to bypass the required minipool backing by deliberately forcing self-transfers between owned addresses via a specific sequence of actions within the Protocol DAO contracts. It has to be noted that this will be subject to a partial burn.</p>
Recommendations	Consider thinking about this issue and whether it exposes a risk with the background that there is usually no backing for minipools required.
Comments / Resolution	<p>Acknowledged, Rocketpool considers this as non-issue and added the following comment:</p> <p>"This is intentional. Legacy staked RPL should not be able to be increased after Saturn. The only way `transferRPL` can be initiated is via the on-chain pDAO system when paying out bonds for successful challenges. There is an amount burnt when bonds are transferred to prevent this sort of exploit. There would be a significant cost to use this method to transfer out legacy staked RPL."</p> <p>The severity has been decreased from low to informational, as indeed such a pathway is possible - but it requires burning a specific share.</p>

## Rewards

### RocketClaimDAO

The **RocketClaimDAO** contract is a simple payment handler which is responsible for setting up payment contracts and executing recurring payments in a permissionless fashion. The following functions are callable by the Protocol DAO contract via proposal executions:

- **newContract**: Creates a completely new contract
- **updateContract**: Modifies the following traits of an existing contract
  - recipient
  - amountPerPeriod
  - periodLength
  - numPeriods
- **spend**: One-time spend to a recipient

This contract has been changed for the Saturn upgrade.

#### Appendix: New Contract Setup

It is possible to create new payment contracts via the DAO governance mechanism. The **RocketDAOProtocolProposals** contract will trigger the **newContract** function which then simply sets up such a contract with the following traits:

- recipient
- amountPerPeriod
- period
- lastPayment
- numPeriods

#### Appendix: Periodical Payouts

Any created contract allows for periodical payouts of **amountPerPeriod** in each defined period. Whenever at least one period has passed, anyone can call the **payOutContracts** function which accounts for the payout then followed by **withdrawBalance** function which then transfers out the accounted payout. Alternatively, the **payoutContractsAndWithdraw** function does both in one call.



## Core Invariants:

INV 1: Only existing contracts can be updated via [updateContract](#)

INV 2: Adding contracts/changing contracts/spending is only callable via PDAO governance proposals

## Privileged Functions

- transferOwnership
- renounceOwnership

Issue_119	Multiple unexpected implications in case of contract update
Severity	Medium
Description	<p>The <code>updateContract</code> function allows to change multiple contract traits via a governance proposal. There are two specific side-effects which can occur from this:</p> <ul style="list-style-type: none"> <li>- If this is called near the end of a period, the period is not paid out proportionally and disregarded</li> <li>- It will always set <code>lastPaymentTime</code> to the end of the previously passed period and <b>not to <code>block.timestamp</code></b>. This is considered as an issue as now the new setup will not be considered from the time where we have updated the contract but rather from the time where the last period has passed, resulting in potential unexpected side-effects</li> </ul> <p>Example:</p> <ul style="list-style-type: none"> <li>- previous <code>amountPerPeriod</code> = 1 ETH</li> <li>- previous period = 2 weeks</li> <li>- Now update 1 day before 2 weeks are reached with the parameters</li> <li>- <code>amountPerPeriod</code> = 5 ETH</li> <li>- <code>newPeriod</code> = 1 week</li> <li>- It would now pay out immediately 5 ETH and a day later again 5 ETH vs the expected 5 ETH in 7 days from now</li> </ul> <p><b>This issue is live in the current deployed version.</b></p>
Recommendations	Consider either refactoring this function to properly reset all traits and prevent potential overlapping or implementing a warning which highlights this behavior.
Comments / Resolution	Acknowledged.

<b>Issue_120</b>	Edge-case if <code>numPeriods</code> is set below <code>periodsPaid</code> during <code>updateContract</code> will result in revert of payout
<b>Severity</b>	<b>Low</b>
<b>Description</b>	Due to incorrect state updates during <code>updateContract</code> , it is possible that <code>numPeriods</code> is below <code>periodsPaid</code> . This results in an underflow during the contract update attempt.
<b>Recommendations</b>	It is non-trivial to fix this as such a proposal must first go through the voting process and while it is in the voting process, <code>numPeriods</code> can still be above <code>periodsPaid</code> but <code>periodsPaid</code> will increase steadily over time. That means, even if such a validation occurs at the proposal level, it may be void once the proposal is executed.
<b>Comments / Resolution</b>	Acknowledged.

<b>Issue_121</b>	Missing ETH claim support
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	Currently the <code>ClaimDAO</code> contract receives ETH via multiple pathways but is not capable of using it.  This is acknowledged with a comment and will be implemented in future versions.
<b>Recommendations</b>	Consider acknowledging this issue.
<b>Comments / Resolution</b>	Acknowledged, in the next upgrade, such a function will be introduced.

Issue_122	Permissionless withdrawal may result in unexpected tax-implications in certain jurisdictions
Severity	Informational
Description	Currently, the fund withdrawal logic is permissionless which sparks concerns over potential tax-implications in the scenario where an address calls this function on behalf of another address while being in specific periods where no profit should be realized.
Recommendations	Consider evaluating such a possibility.
Comments / Resolution	Acknowledged.

## RocketMerkleDistributorMainnet

The [RocketMerkleDistributorMainnet](#) contract receives tokens via the [RocketRewardsPool](#) in the scenario of a successfully executed snapshot. This will then trigger the [relayRewards](#) function which sets the merkle root for the `rewardIndex` and accounts for the received RPL tokens and native ETH by depositing these into the [RocketVault](#) contract.

Users can then claim their corresponding reward share either via the [claim](#) or via the [claimAndStake](#) function. The latter function simply stakes the RPL within the [NodeStaking](#) contract instead of executing a direct withdrawal.

**This contract has been changed for the Saturn upgrade.**

### Appendix: Claim Process:

Below we will describe the claim process with a direct withdrawal and the claim process via staking - step by step.

#### Direct Withdrawal:

- loop over all claims
  - aggregate `totalAmountRPL`
  - aggregate `totalSmoothingPoolETH`
  - aggregate `totalAmountVoterETH`
- withdraw `totalAmountRPL` from [RocketVault](#)
- withdraw `totalAmountSmoothingPoolETH` and `totalAmountVoterETH` from [RocketVault](#)
- transfer `totalAmountSmoothingPoolETH` to `withdrawalAddress`
- transfer `totalAmountVoterETH` to `RPLWithdrawalAddress`
- optionally - if transfers revert - account for failed transfers and allow later claim

#### Staking:

- loop over all claims
  - aggregate `totalAmountRPL`
  - aggregate `totalSmoothingPoolETH`
  - aggregate `totalAmountVoterETH`
- ensure `stakeAmount` is  $\leq$  `totalAmountRPL`
- calculate remaining [`totalAmountRPL` - `stakeAmount`]
- withdraw remaining to `RPLWithdrawalAddress`

- withdraw totalAmountRPL from RocketVault
- withdraw totalAmountSmoothingPoolETH and totalAmountVoterETH from RocketVault
- transfer totalAmountSmoothingPoolETH to withdrawalAddress
- transfer totalAmountVoterETH to RPLWithdrawalAddress
- optionally - if transfers revert - account for failed transfers and allow later claim
- stake RPL on the NodeStaking contract for the nodeAddress

## Appendix: Merkle Proof Verification

As with any other merkle root based claiming process, it is required to mark leaves as used in the effort to prevent double-spending. Below follows an exhaustive explanation of the verification process.

Each leaf consists of:

- nodeAddress
- network
- amountRPL
- amountSmoothingPoolETH
- amountVoterETH

## Replay Protection:

The replay protection within `_verifyClaim` was implemented against the duplicate usage of the same leaf. Instead of simply mapping `keccak256[nodeAddress, rewardIndex]` to a boolean, a gas-optimized approach was chosen which leverages bitmap partitioning. The flow for this is as follows:

- Determination of `indexWordIndex` via dividing the `rewardIndex` by 256:
  - $indexWordIndex = rewardIndex / 256$
  - This has the effect that the first 256 [0-255] rewardIndexes share the same `indexWordIndex` [0] and the subsequent 256 [256-511] share the next `indexWordIndex` [1]
- Fetching of `claimedWord` via `claimedWordKey`:
  - `getUint[keccak256[abi.encodePacked['rewards.interval.claimed', _nodeAddress, indexWordIndex]]]`
  - This has the effect that each node has its own unique "`claimedWord`" which keeps track claims for each 256 batch of reward indexes
- Loop over each Claim array
  - Determination of `indexBitIndex`:
    - $indexBitIndex = rewardIndex \% 256$

- This has the effect that each rewardIndex has its own unique indexBitIndex, which points to a specific slot within the indexWordIndex
- Creation of mask
  - $\text{mask} = 1 \ll \text{indexBitIndex}$
  - This mask is then used to check whether the claimedWord has already marked a claim on this specific slot
- Mark rewardIndex as claimed:
  - $\text{claimedWord} = \text{claimedWord} | [1 \ll \text{indexBitIndex}]$
  - This has the effect that this specific slot is considered as claimed and cannot be re-used
- Bonus: In the scenario where within the array operation, a rewardIndex is considered which is not yielding the same indexWordIndex of the previous rewardIndex, the previous operation results are stored and the new indexWordIndex is used for continuation.

#### properties:

- the first 255 rewardIndexes share the same indexWordIndex/claimedWordKey
- indexBitIndex for the first 255 rewardIndexes will be from 0 to 255
- This means that each rewardIndex has its own place within the shared claimedWordKey
- mask is created by shifting 1 for each rewardIndex [from 0 to 255]
- The claimedWord is adjusted after each reward claim for  $[1 \ll \text{rewardIndex}]$

#### Core Invariants:

INV 1: Each leaf can only be claimed once

INV 2: stakeAmount must not be larger than totalAmountRPL

INV 3: Unsuccessful transfers must be accounted to rewards.eth.balance

INV 4: Only RPLWithdrawalAddress/nodeAddress or withdrawalAddress can invoke the claim/claimAndStake functions - depending on the control-flow

#### Privileged Functions

- relayRewards

Issue_123	Unclaimable rewards post-update due to mismatch of leaf
Severity	High
Description	<p>Rocket Pool distributes multiple ETH/RPL reward components to node operators through a <a href="#">MerkleDistributor</a>.</p> <p>Prior to the saturn upgrade, the distributor's Merkle leaf committed to four fields:</p> <ul style="list-style-type: none"> <li>- nodeAddress</li> <li>- network</li> <li>- amountRPL,</li> <li>- amountETH</li> </ul> <p>proofs were generated off that exact tuple.</p> <p>The upgraded distributor refactors claim inputs to a <a href="#">Claim</a> struct whose leaf commits to a different field set:</p> <ul style="list-style-type: none"> <li>- nodeAddress</li> <li>- network</li> <li>- amountRPL</li> <li>- amountSmoothingPoolETH</li> <li>- amountVoterETH.</li> </ul> <p>Operators may still have unclaimed legacy rewards (built into an <i>old</i> Merkle root and leaf schema) when the <i>new</i> distributor goes live.</p> <p>The upgrade changes the Merkle leaf schema, but the new verifier only accepts proofs for the new schema. A Merkle proof is specific to the exact leaf bytestring used when the tree was built; inserting or re-ordering fields (e.g., replacing amountETH with two components amountSmoothingPoolETH/amountVoterETH) changes the leaf hash and therefore the entire branch.</p> <p>Consequently, a user holding an old proof cannot satisfy the new validation function: the recomputed leaf (using the new struct) will not match the old proof's path, and the on-chain check fails. The root cause is a backward-incompatibility in the commitment format</p>

	<p>without a compatibility layer [dual validation] or a continued legacy distributor for outstanding claims.</p> <p>Legacy, unclaimed rewards become inaccessible after the upgrade: users cannot prove them to the new distributor, and the old distributor is [presumably] retired.</p> <p>This is a funds-availability failure [economic loss/lock] for operators who did not claim before the cutover. In practical terms, anyone relying on the upgrade to “just work later” will discover that their historical entitlement cannot be claimed, absent governance/manual remediation.</p>
<b>Recommendations</b>	Consider keep the legacy distributor contract and root live for an overlap window so users can finish claiming old entitlements. It is also possible to refactor the MerkleDistributor contract in an effort to support historic proofs. However, this would require an entire audit of the refactored control-flow in an effort to make sure that no unexpected side-effects occur.
<b>Comments / Resolution</b>	<p>Open. This has been resolved in commit <a href="https://github.com/rocketpool/rocketpool/commit/9af2624a462a779362f19a0ad20d7bf6738e3f01">https://github.com/rocketpool/rocketpool/commit/9af2624a462a779362f19a0ad20d7bf6738e3f01</a>.</p> <p>However, instead of keeping the old distributor live, the claim function was fully refactored to support old and new claims at the same time. All control-flows need to be fully validated in an effort to confirm correct behavior, including the mechanism of how the old rewards were supposed to be claimed in the pre Saturn commit.</p>
<b>Resolution 2 / Follow-Up</b>	Resolved, the modular <code>claimAndStake</code> function now allows old and new claims depending on the version.

Issue_124	Edge-case in <code>claimOutstandingETH</code> may result in permanently locked funds
Severity	Medium
Description	<p>In Rocket Pool, a node operator may change their withdrawal address/RPL withdrawal address at will. Multiple subsystems send ETH to this address.</p> <p>The MerkleDistributor's <code>_claimAndStake</code> attempts to pay the node's <i>current</i> withdrawal/RPL withdrawal address using a 10,000-gas call as a defensive measure against gas-hungry fallback logic on smart-contract wallets. If that low-gas send fails, the contract "credits back" the amount for later claiming.</p> <p>Operators rely on the general property that, by changing the withdrawal address, they can later receive funds safely with a standard gas-forwarding call.</p> <p>In this distributor, the "credit back" on send failure is recorded against the <i>withdrawal/RPL withdrawal address itself</i>, and later only that exact address is authorized to claim and/or is used again as the receiver. If the credited address is a non-payable contract or otherwise incapable of receiving ETH, then:</p> <ol style="list-style-type: none"> <li>1. <code>_claimAndStake</code> fails the 10,000-gas send and credits the balance to that address</li> <li>2. a subsequent <code>claim()</code> path still requires [or pays to] the same address, and the call fails again</li> <li>3. changing the node's withdrawal address later does not move the credit and does not change who can claim</li> <li>4. the previously credited ETH is permanently locked behind an address that can never receive it.</li> </ol>

	<p>This diverges from other Rocket Pool flows where “unclaimed” amounts are tracked per node and later paid to the current withdrawal address with normal gas, allowing the operator to fix reception issues by simply updating their address.</p> <p>The outcome is funds loss in practice (permanent lockup) for the affected node’s distributor rewards. While the scenario is an edge-case, it is plausible: operators may temporarily point the withdrawal address to a smart-contract wallet (for custody, tax timing, or testing) that has either a non-payable fallback or strict logic causing the 10,000-gas send to fail.</p> <p>In all other protocol flows the operator can recover by later switching the withdrawal address; here, the credit is hard-allocated to the old address and can’t follow the node, so subsequent claims with unlimited gas still target/authorize the wrong endpoint.</p>
<b>Recommendations</b>	The simplest mitigation is to allow initiating a claim to an explicit recipient (pull-with-recipient), enabling safe redirection without compromising authorization.
<b>Comments / Resolution</b>	Acknowledged.

## RocketRewardsPool

The [RocketRewardsPool](#) contract allows trusted nodes to submit a reward snapshot which has the following traits:

- rewardIndex
- executionBlock
- consensusBlock
- merkleRoot
- intervalsPassed
- smoothingPoolETH
- treasuryRPL
- treasuryETH
- userETH
- trustedNodeRPL
- nodeRPL
- nodeETH

The logic follows the exact same procedure as all other trusted node submissions: a snapshot can be executed once the consensus threshold has been passed. It is important to mention that the actual derivation of the snapshot data is unknown and it is expected that the consensus threshold is a sufficient safeguard to ensure this data is always accurate and matching

**This contract has been changed for the Saturn upgrade.**

### Appendix: Reward Seeding

The [RewardsPool](#) has different methods of receiving RPL and ETH which is then allocated within the RocketVault contract and can be withdrawn whenever a snapshot is executed.

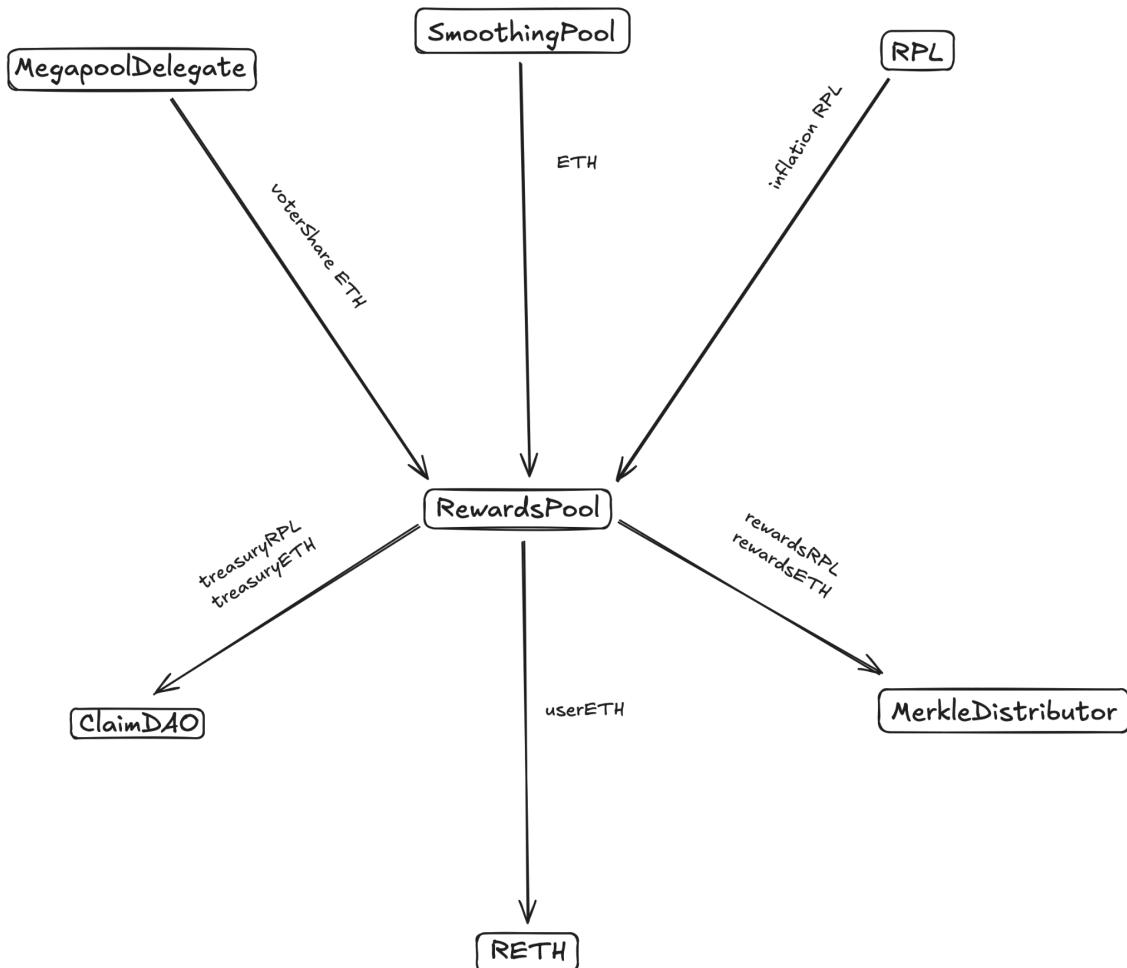
ETH:

- Megapool reward distribution: voterAmount
- SmoothingPool claims

RPL:

- Via simple periodical RPL inflation

We have illustrated this behavior in the graphic below:



#### Core Invariants:

INV 1: `_executeRewardSnapshot` must increment `rewards.snapshot.index`

INV 2: RPL amount for reward snapshot [cannot be larger than pending RPL rewards](#)

INV 3: ETH amount for reward snapshot [cannot be larger than smoothingPoolBalance and voterShare](#)

INV 4: RewardsPool must receive ETH from MegapoolDelegate (`voterShare`)

INV 5: RewardsPool must receive RPL inflation

INV 6: `_submission.nodeRPL.length` must be zero (due to 1 network)

## Privileged Functions

- none

<b>Issue_125</b>	Inefficient rewardsRPL distribution
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>The RPL token distribution happens via a withdrawal from the <a href="#">RocketVault</a> just to then being relayed to the <a href="#">MerkleDistributor</a> and re-deposited to the vault.</p> <p>One can simply flow the transferToken function to reduce a few steps and save some gas.</p>
<b>Recommendations</b>	We recommend acknowledging this issue in an effort to prevent further code changes.
<b>Comments / Resolution</b>	Acknowledged.

Issue_126	Potentially incorrect off-chain calculation in case of user-triggered <code>depositVoterShare</code> call
Severity	Informational
Description	<p>The <code>RocketRewardsPool</code> exposes <code>depositVoterShare()</code> (payable), which immediately forwards received ETH into <code>RocketVault</code> under the pool's own account.</p> <p>Off-chain, trusted nodes collate a <code>RewardSubmission</code> and decide how much ETH (from Smoothing Pool + voter share in the Vault) to distribute in the next snapshot.</p> <p>The pool's public view functions (<code>getEthBalance()</code> / <code>getPendingVoterShare()</code>) return the Vault balance attributed to the pool, and <code>getRPLBalance()</code> returns the Vault's RPL balance for the pool.</p> <p>Because <code>depositVoterShare()</code> is permissionless, any address can increase the pool's voter-share Vault balance. The on-chain views then report a higher "pending voter share" regardless of the deposit's provenance.</p> <p>However, the protocol does not tie snapshot construction to those view values: trusted nodes are only required to ensure <code>totalRewardsETH &lt;= smoothingPoolBalance + voterShare</code>. They can exclude arbitrary deposits from the next <code>RewardSubmission</code>.</p> <p>Thus, the system can present inflated, view-only "pending" ETH that may not be credited in the very next cycle if the off-chain policy ignores such deposits.</p> <p>User-visible impact (ETH): <code>getPendingVoterShare()</code> / <code>getEthBalance()</code> can overstate what will be distributed in the next snapshot. This can mislead operators/UIs and complicate monitoring (apparent "pending" funds that are not actually allocated by the next <code>RewardSubmission</code>). Deposited ETH is not lost, it remains in the Vault under the pool's account and can be distributed in a later cycle if the quorum chooses, but there's no on-</p>

	chain guarantee it will be.
<b>Recommendations</b>	We recommend implementing strong off-chain mechanics to handle such cases.
<b>Comments / Resolution</b>	Acknowledged.

<b>Issue_127</b>	Fallback ETH is never accounted for
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	Currently, any ETH received via fallback is not accounted for as there is no method to deposit such into the <a href="#">RocketVault</a> contract and also no native balance check is executed at any time.
<b>Recommendations</b>	Consider acknowledging this issue, as it is not expected that any ETH is received outside the closed cycle.  We do not recommend a code change.
<b>Comments / Resolution</b>	Acknowledged.

## RocketSmoothingPool

The [RocketSmoothingPool](#) contract is a simple custodian contract for bundled MEV fees/rewards from validators. These funds are simply stored and withdrawn upon requirement from the RewardsPool contract.

**This contract has not been changed for the Saturn upgrade.**

### Privileged Functions

- withdrawEther

No issues found.

## Token

### RocketTokenRETH

The `RocketTokenRETH` contract is the yield bearing token which represents the underlying ETH within the `RocketPool` system. It implements a simple mint and burn function which allows users to exchange their ETH to RETH and vice-versa based on the current exchange rate. Minting is solely allowed via the `DepositPool` which incurs a fee and further safety checks. The `RETH` contract can be considered as a receipt for underlying ETH which is considered to increase in ETH value over time due to profit from ETH staking.

The contract exposes the ETH  $\leftrightarrow$  RETH exchange rate and also the inverse RETH  $\leftrightarrow$  ETH exchange rate. These are based on `network.balance.reth.supply` and `network.balance.total`. Both values are stored within the `NetworkBalances` contract and are subject to change via trusted node majority submission. Any on-chain state changes will not be reflected on the exchange rate until `NetworkBalances.updateBalances` is invoked.

It has to be noted that burns are constrained by the current ETH balance within the RETH contract with an additional buffer balance in the `DepositPool` contract for “excess balance”.

The contract targets to only keep the target collateralization (default 10%) of ETH as balance.

**This contract has not been changed for the Saturn upgrade.**

**Core Invariants:**

INV 1: Contract must only keep the target collateralization as balance

INV 2: Burn is allowed to consume excess balance in DepositPool (balance above capacity)

INV 3: Mint must use the current ETH  $\rightarrow$  RETH exchange rate

INV 4: Burn must use the current RETH  $\rightarrow$  ETH exchange rate

INV 5: burn can only happen if balance is covered by RETH contract balance [and DepositPool excess balance](#)

## Privileged Functions

- depositExcess
- mint
- depositExcessCollateral

<b>Issue_128</b>	Usage of transfer is sub-optimal
<b>Severity</b>	Low
<b>Description</b>	<p>The <code>burn</code> function transfers ETH via the simple transfer mechanism which forwards 2300 gas only.</p> <p>This can be a constraint for smart contract wallets that execute logic upon fallback - as it can simply revert.</p>
<b>Recommendations</b>	Consider using <code>.call</code> instead of <code>.transfer</code>
<b>Comments / Resolution</b>	Acknowledged.

<b>Issue_129</b>	Unused limitation within <code>_beforeTokenTransfer</code>
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>The mentioned function exposes a limitation which prevents transfers until the delay time after <code>user.deposit.block</code> has passed.</p> <p>However, in the current architecture, this value is actually never set, making this limitation void.</p>
<b>Recommendations</b>	Since we assume that this was used in the past and is now deprecated, we simply recommend keeping this check as-is, in an effort to prevent redundant code changes.
<b>Comments / Resolution</b>	Acknowledged, it was used in a previous iteration.

<b>Issue_130</b>	Reliance on <code>executeBalance</code> can result in MEV
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>As already explained in the contract description, the exchange rate relies on the state within the <code>NetworkBalances</code> contract. This opens up the path for MEV because users can already calculate the new exchange rate or frontrun the <code>updateBalance</code> call which then allows them to immediately withdraw again for a profit.</p> <p>This issue is only rated as informational due to the fee which is implemented. Furthermore, the RocketPool protocol is already very large which makes such jump-wise increases less likely.</p>
<b>Recommendations</b>	Consider keeping this in mind.
<b>Comments / Resolution</b>	Acknowledged.

## RocketTokenRPL

The [RocketTokenRPL](#) contract is the protocol token which is distributed on several occasions and implements an inflation mechanism.

The contract implements discrete, per-interval compounding of total supply, multiplying the supply by a DAO-set per-day factor for each full day elapsed.

The design purposefully batches issuance into whole-day increments, which simplifies timekeeping and allows infrequent callers to mint for multiple elapsed intervals in one go.

**This contract has not been changed for the Saturn upgrade.**

### Core Invariants:

INV 1: inflationCalcTime must point to the block.timestamp when the next inflation interval is triggered

INV 2: inflationMintTokens must deposit newly minted tokens into the RocketVault on behalf of the RewardsPool

INV 3: swapTokens must honor 1:1 exchange and increase totalSwappedRPL by the swapped amount

### Privileged Functions

- none

<b>Issue_131</b>	Unbounded loop can lead to permanent DoS in times of inactivity
<b>Severity</b>	<b>Low</b>
<b>Description</b>	<p>The <code>inflationMintTokens</code> function triggers <code>_inflationCalculate</code> which loops over each period (1 day) and accumulates the pending tokens.</p> <p>In a scenario of severe inactivity it can happen that this function call runs out of gas, even worse, exceeds the block gas limit and results in a permanent DoS of this functionality. This will essentially partially shut down the protocol.</p> <p>Due to the fact that the likelihood of this happening is very small, we have rated this issue only as low severity.</p>
<b>Recommendations</b>	Since this contract is not meant to be upgraded in the saturn upgrade, we recommend acknowledging this issue. Even if this issue ever occurs, governance can simply upgrade the implementation and implement a solution which exhausts a certain amount of intervals without running out of gas.
<b>Comments / Resolution</b>	Acknowledged.

<b>Issue_132</b>	Potentially non-existing RPL tokens for swapping purposes
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>The <code>swapTokens</code> function allows users to swap their old RPL token to the new one, while a simple transfer to the user with the new token is being executed.</p> <p>In the current instance of the contract, there is no state where RPL tokens are sitting idle.</p> <p>However, we highly assume that this was part of a previous upgrade and therefore, there was a mechanism that ensures that the contract now holds sufficient idle tokens to honor all swaps.</p>
<b>Recommendations</b>	Consider clarifying this scenario.
<b>Comments / Resolution</b>	Acknowledged - in fact this was handled in an earlier iteration.

# Upgrade

## RocketUpgradeOneDotFour

The [RocketUpgradeOneDotFour](#) contract is responsible for setting the correct upgrade traits. This not only includes the addition of new contracts such as contracts which are tied to the Megapool module but also includes the correct upgrade of contracts which are changed.

Furthermore, contracts that require initialization are being initialized and multiple new variables are initialized.

The [\\_upgradeContract](#) and [\\_addContract](#) functions match with the implementation within the DAO module.

It has to be noted that it would be theoretically also possible to handle all these settings via governance proposals.

### Appendix: Upgrade Overview

Please note that this audit does not specifically cover any migration process whatsoever. However, due to the fact that the [RocketUpgradeOneDotFour](#) contract is in scope, it is required to review the correctness of new contract initialization and initialization of variables for contracts that have received an upgrade.

#### New Contracts:

- RocketDAOProtocolSettingsMegapool [DONE]
- RocketDAOSecurityUpgrade [DONE]
- RocketMegapoolDelegate [DONE]
- RocketMegapoolDelegateBase [DONE]
- RocketMegapoolFactory [DONE]
- RocketMegapoolManager [DONE]
- RocketMegapoolPenalties [MISSING]
- RocketMegapoolProxy [DONE]
- RocketMegapoolStorageLayout [DONE]
- RocketNetworkRevenues [DONE]

## Changed Contracts:

- RocketDAOUpgrade: <https://www.diffchecker.com/kQkyDujI/> [DONE]
- RocketDAOProtocolSettingsDeposit: <https://www.diffchecker.com/CRROkqXA/> [DONE]
- RocketDAOProtocolSettingsMinipool: <https://www.diffchecker.com/nmHKWwAi/> [DONE]
- RocketDAOProtocolSettingsNetwork: <https://www.diffchecker.com/vrU7cVhT/> [DONE]
- RocketDAOProtocolSettingsNode: <https://www.diffchecker.com/U6XXRfLU/> [DONE]
- RocketDAOProtocolSettingsSecurity: <https://www.diffchecker.com/3UocUCLQ/> [DONE]
- RocketDAOProtocol: <https://www.diffchecker.com/7eCWTMn1/> [DONE]
- RocketDAOProtocolProposals: <https://www.diffchecker.com/kKNmtkWp/> [DONE]
- RocketDAOSecurityProposals: <https://www.diffchecker.com/FnQ7n7AI/> [DONE]
- RocketDepositPool: <https://www.diffchecker.com/Uuib8r2N/> [DONE]
- RocketMinipoolBondReducer: <https://www.diffchecker.com/owCx9VfF/> [DONE]
- RocketMinipoolManager: <https://www.diffchecker.com/EvGJOSGM/> [MISSING]
- RocketNetworkBalances: <https://www.diffchecker.com/RUiDysbW/> [DONE]
- RocketNetworkPenalties: <https://www.diffchecker.com/pbpgkhQz/> [DONE]
- RocketNetworkSnapshots: <https://www.diffchecker.com/6632cjda/> [DONE]
- RocketNetworkVoting: <https://www.diffchecker.com/8YdQ3J3B/> [DONE]
- RocketNodeDeposit: <https://www.diffchecker.com/hMgodudi/> [DONE]
- RocketNodeDistributorDelegate: <https://www.diffchecker.com/PfovW4kp/> [DONE]
- RocketNodeDistributorStorageLayout: <https://www.diffchecker.com/64RgdBa3/> [DONE]
- RocketNodeManager: <https://www.diffchecker.com/nDyEdQhf/> [DONE]
- RocketNodeStaking: <https://www.diffchecker.com/4yqyeAzx/> [DONE]
- RocketClaimDAO: <https://www.diffchecker.com/S34CSLrh/> [DONE]
- RocketMerkleDistributorMainnet: <https://www.diffchecker.com/IhCjc65f/> [DONE]
- RocketRewardsPool: <https://www.diffchecker.com/YBENhd5V/>
- RocketBase: <https://www.diffchecker.com/ARy8ac20/>

## Privileged Functions

- transferOwnership
- renounceOwnership

Issue_133	Lack of contract add/upgrade
Severity	Medium
Description	<p>The <code>RocketMegapoolPenalties</code> contract is newly added and the <code>RocketMinipoolManager</code> is changed. Neither one is added/upgraded.</p> <p>The impact is that there is no penalty possibility for Megapools until a governance vote is passed which adds the <code>RocketMegapoolPenalties</code> contract.</p>
Recommendations	Consider adding the <code>RocketMegapoolPenalties</code> contract and upgrading the <code>RocketMinipoolManager</code> .
Comments / Resolution	<p>Resolved, within the provided fix commit:</p> <p><a href="https://github.com/rocket-pool/rocketpool/commit/ca3d8ebc1fc2cfa87d8e9538215938019ca53bd">https://github.com/rocket-pool/rocketpool/commit/ca3d8ebc1fc2cfa87d8e9538215938019ca53bd</a></p> <p>Only the <code>RocketMegapoolPenalties</code> contract has been added. The <code>RocketMinipoolManager</code> contract is not changed.</p> <p>However, in the final provided commit, the <code>RocketMinipoolManager</code> is upgraded:</p> <pre>_upgradeContract("rocketMinipoolManager", addressesB[12], abisB[12]);</pre>

Issue_134	Incorrect ABI setting
Severity	Informational
Description	The ABI for <code>BlockRoots</code> and <code>BeaconStateVerifier</code> are set incorrectly. This will however only have informational consequences.
Recommendations	Consider fixing the ABI settings.
Comments / Resolution	Resolved.

# Util

## AddressQueueStorage

The [AddressQueueStorage](#) contract is a helper contract which is solely used for managing the Minipool queue. It exposes standard functionality such as adding and removing Minipoools to the queue as well as multiple different view functions which display the current queue status

**This contract has not been changed for the Saturn upgrade.**

### Appendix: Enqueue/Dequeue

Minipools are enqueued within the [MinipoolQueue](#) contract whenever a new Minipool is created. However, this specific mechanism is outdated after Saturn, as Minipools cannot be created anymore. Contrary to that, Minipools can still be dequeued whenever an assignment happens and a Minipool is removed from the queue.

The process of enqueueing is as follows:

- Fetch the index (default zero)
  - `getUint[keccak256[abi.encodePacked(_key, ".end")]]`
- Tie address to index:
  - `setAddress[keccak256[abi.encodePacked(_key, ".item", index)], _value]`
- Tie index+1 to address:
  - `setUint[keccak256[abi.encodePacked(_key, ".index", _value)], index.add[1]]`
- Increase index and save it:
  - `index = index.add[1]`
  - `setUint[keccak256[abi.encodePacked(_key, ".end")], index]`

The process of dequeuing is as follows:

- Fetch the start (default zero):
  - `start = getUint[keccak256[abi.encodePacked(_key, ".start")]]`
- Fetch Minipool address at start:
  - `getAddress[keccak256[abi.encodePacked(_key, ".item", start)]]`
- Increase start for next assignment
  - `start = start.add[1]`
- Reset index to address:
  - `setUint[keccak256[abi.encodePacked(_key, ".index", item)], 0]`

- Set increased start:
  - `setUint[keccak256[abi.encodePacked(_key, ".start")], start]`

The removal of an item is not treated within this audit because it is not triggered within the current architecture. Minipools cannot be removed from the queue.

#### Core Invariants:

INV 1: No address must appear more than once

INV 2:  $\text{length} = [\text{end} - \text{start}] \bmod \text{capacity}$  and must equal the number of enqueued-but-not-dequeued items

INV 3: If  $.index[x] = i+1$ , then the slot at  $i$  must store  $x$

#### Privileged Functions

- none

<b>Issue_135</b>	Address remains tied to an index after <code>dequeue</code>
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>During the <code>dequeue</code> function, the address to index setting:</p> <pre>setAddress[keccak256[abi.encodePacked(_key, ".item", index)], _value];</pre> <p>is not removed. While this has no impact as it is not actively used, it can confuse third-party reviewers.</p> <p>The same is true for <code>removeItem</code> with the last index.</p>
<b>Recommendations</b>	Our recommendation would be to simply reset the ties during the <code>dequeue</code> function. However, since there is no planned upgrade for this contract, we recommend simply acknowledging this issue to prevent safety concerns during upgrades. Furthermore, it will result in an inconsistency between past and future storage content.
<b>Comments / Resolution</b>	Acknowledged.

## AddressSetStorage

The [AddressSetStorage](#) contract is a helper contract which is used for managing different versions of Minipools and different states. These are the following:

- `minipools.index`: Amount of created and not scrubbed/dissolved Minipools
- `node.minipools.index`: Amount of created and not scrubbed/dissolved Minipools, tied to a node
- `minipools.vacant.index`: Amount of vacant Minipools that are not yet promoted or dissolved
- `node.minipools.validating.index`: Amount of created Minipools, never decreased, tied to a node

It furthermore keeps track of all registered nodes:

- `nodes.index`: Amount of registered nodes

And also keeps track of Trusted Nodes and DAO Security Members.

It follows the same add/remove mechanics as the [AddressQueueStorage](#) contract.

**This contract has been changed for the Saturn upgrade.**

Core Invariants:

INV 1: index is 1-based

INV 2: There are count addresses with non-zero .index

Privileged Functions

- none

No issues found.

## BeaconStateVerifier

The **BeaconStateVerifier** is newly implemented during the Saturn upgrade and allows Megapools to verify **WithdrawalProofs** and **ValidatorProofs** against the beacon state, which allows the usage of Megapools without a scrub period (Minipools).

It essentially exposes two functions:

- a] verifyValidator
- b] verifyWithdrawal

These functions have a single proof as input parameter and are then using the SSZ library to compute the root from the proof and compare it against the valid root for the specific slot via the **BlockRoots** contract which fetches the root from the **BeaconRoots** contract.

## Appendix: ValidatorProof Verification

### Path Creation

The **ValidatorProof** verification process reconstructs a Beacon Block Root with a provided **ValidatorProof** and **witnesses**. This reconstructed root is then compared with the valid Beacon Block Root that is fetched from the **BeaconRoots** contract.

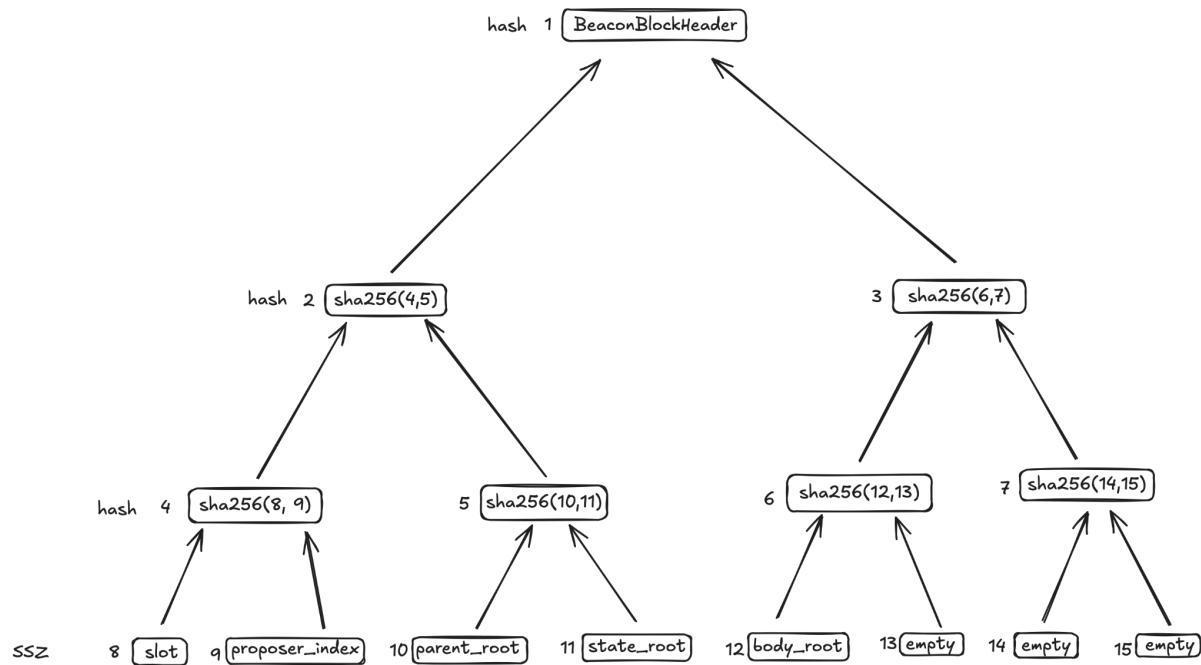
To understand the process of the reconstruction, we will provide a walk-through example which starts with crafting the Path from the Beacon Block Header down to the corresponding validator - step by step.

The Beacon Block Root is the SSZ Merkle Root of a **BeaconBlockHeader** container:

#### BeaconBlockHeader:

- [0] slot: Slot
- [1] proposer\_index: ValidatorIndex
- [2] parent\_root: Root
- [3] **state\_root: Root**
- [4] body\_root: Root
- [5] empty
- [6] empty
- [7] empty

These are 8 padded leaves [SSZ serialized] and are built upwards to the root via sha256:



Within the `pathBeaconBlockHeaderToStateRoot` function, the function fetches the path via `SSZ.from[3,3]`. This relates to:

- depth = 3 - as the leaves are at depth 3
- tail bits = 3 [3rd index at the bottom of the tree] [binary: 011] - explaining the path downwards [left/right/right]

Which correctly points to `state_root`. The return value as path = 771 [0x0303]. If now the `tolIndex` function is used, path = 771 will be converted to gindex = 11.

The following witnesses which are required are: [parent\_root, gindex 4, gindex 3] which is then used to build up to the header.

After the Path from the Beacon Block Header to the Beacon State Root was crafted, the next step is to craft the Path from the **Beacon State to the corresponding validator for the provided validator index**.

The **BeaconState** is a large container with multiple different data types:

#### BeaconState:

- [0] genesis\_time: uint
- [1] genesisValidatorsRoot: Root
- [2] slot: Slot
- [3] fork: Fork
- ....
- [11] validators: List
- ...
- [36] pendingConsolidations

As we can see, on the 11th index is the validators list, which in itself, contains all **Validator** containers. The **BeaconState** Tree is built up as follows:



Within the **pathBeaconStateToValidator** function, the function fetches the path via **SSZ.from[11,6]**. This relates to:

- depth = 6
- tail bits = 11 (11th index at the bottom of the tree)

Similar to the previous build up, the witnesses are required to build up the tree, which is index 10 (`eth1_deposit_index`) and all sibling hashes up to the root.

The Validator container is data type which is used within the list of the 11th index of the **BeaconState** container

**Validator:**

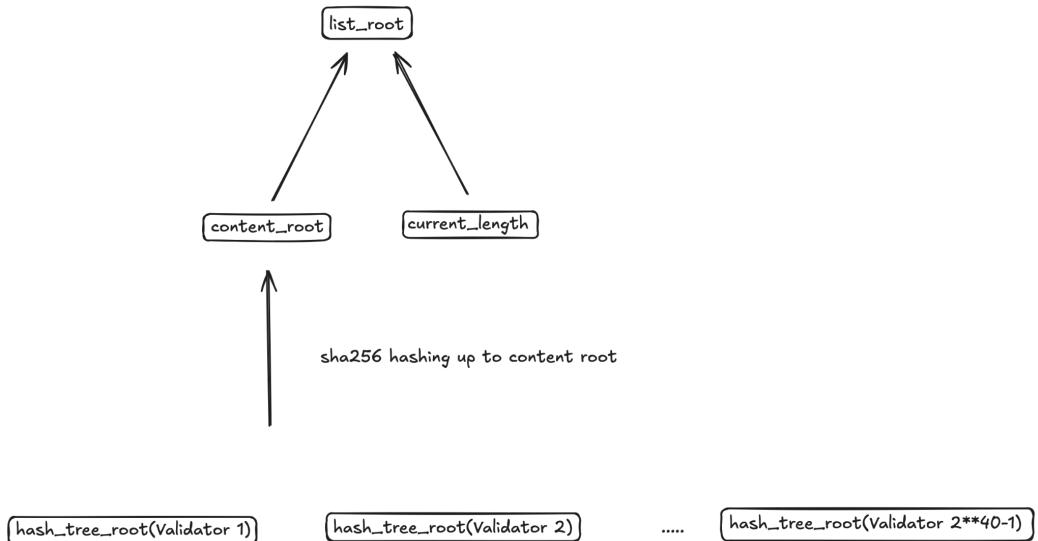
- pubkey: BLSPubkey
- withdrawal\_credentials: Bytes32
- effective\_balance: Gwei
- slashed: boolean
- activation\_eligibility\_epoch: Epoch
- activation\_epoch: Epoch
- exit\_epoch: Epoch
- withdrawable\_epoch: Epoch

The next step is to go into the validators List within the Beacon Root, which we just crafted the path for above. The validator list can have up to  $2^{40}$  validators and each validator is stored as a leaf only:

```
A = SHA256[ merkleisePubkey[pubkey], withdrawal_credentials ]  
B = SHA256[ toLE[effective_balance], toLE[slashed ? 1: 0] ]  
C = SHA256[ toLE[activation_eligibility_epoch], toLE[activation_epoch] ]  
D = SHA256[ toLE[exit_epoch], toLE[withdrawable_epoch] ]  
root = SHA256[ SHA256[A,B], SHA256[C,D] ]
```

These leaves are then sequentially stored within the validators list, the very first created validator will be at index 0 and will never be changed. The only change will be within the unique variables of the validator.

The validators data itself is a List with up to  $2^{40}$  validators. The `list_root` `SHA256[content_root, toLittleEndian[current_length]]` whereas `content_root` is the merkle root for the validators in the list. Each validator in the list is hashed as a merkle root itself and sits at depth = 41 of the `list_root`, sequentially added for each new validator creation. It is important that a List does not only represent the content but also the length:



With this, the path has been fully created. The next step is now to recompute the block root via the provided witnesses, created path and merkle root of the validator.

## TLDR Path:

```
BeaconBlockHeader -> BeaconState  
BeaconState -> validators [List]  
validators toList -> validatorIndex ([leaf])
```

## Root Computation

The first step is to create the SSZ merkle root of the validator via the `merkleiseValidator` function. This simply represents the leaf for the `validatorIndex`. Together with this leaf, the `gindex` which is derived from the overall path (not to be confused with index for each separate tree) and the `witnesses`, the root is now computed within the `restoreMerkleRoot` function which simply hashes the siblings up to the final root that represents the beacon block root. Once the computation is finalized, a simple check is executed whether the recomputed root matches the block root from the `BeaconRoots` contract. We will create a separate appendix for `restoreMerkleRoot`.

## Appendix: WithdrawalProof Verification

The verification of the `WithdrawalProof` follows a different path, while the first part of the path is identical to the `ValidatorProof` verification. The path is as follows:

**BeaconBlockHeader to BeaconState:** This is exactly the same as in the previous appendix, it will simply fetch `gindex = 11 [state_root]`. For illustration we refer to the previous appendix (`SSZ.from[3,3]`)

For the `BeaconState` to `BlockRoot` path, there are two different scenarios, which are dependent on the fact whether the proof is a historical proof or not. The determination for this is as follows:

not historical: `withdrawal.slot + SLOTS_PER_HISTORICAL_ROOT >= proof.slot` (target is within the rolling window)

In that scenario, the path needs to be crafted to `BeaconState.block_roots[withdrawal.slot % SLOTS_PER_HISTORICAL_ROOT]`

historical: `withdrawal.slot + SLOTS_PER_HISTORICAL_ROOT < proof.slot` (target older than the rolling window)

In that scenario, the path needs to be crafted to `BeaconState.historical_summaries[...].block_summary_root[...]`

As one can see, if the slot of the withdrawal is not within `SLOTS_PER_HISTORICAL_ROOT` of the proof slot (e.g. it is too much in the past), then it is considered as a historical proof. In real terms, if the withdrawal slot is more than 8192 slots behind the proof slot, then it will be considered a historical proof.

At this stage, we will simply split this explanation and first continue with the non historic proof until the full path is built and then only after that, continue with the historic proof.

### BeaconState to BlockRoot via non-historical proof

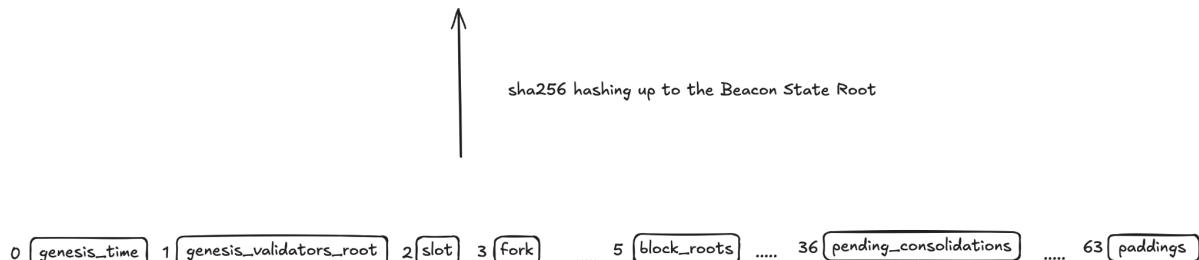
The `block_roots` is simply a Vector within the `BeaconState` container:

### BeaconState:

- [0] genesis\_time: uint64
- [1] genesisValidatorsRoot: Root
- [2] slot: Slot
- [3] fork: Fork
- [4] latestBlockHeader: BeaconBlockHeader
- **[5] block\_roots: Vector**
- .....

As one can see, it is simply the 5th index within the `BeaconState` container, which means we need to apply `SSZ.from[5, 6]` which translates into:

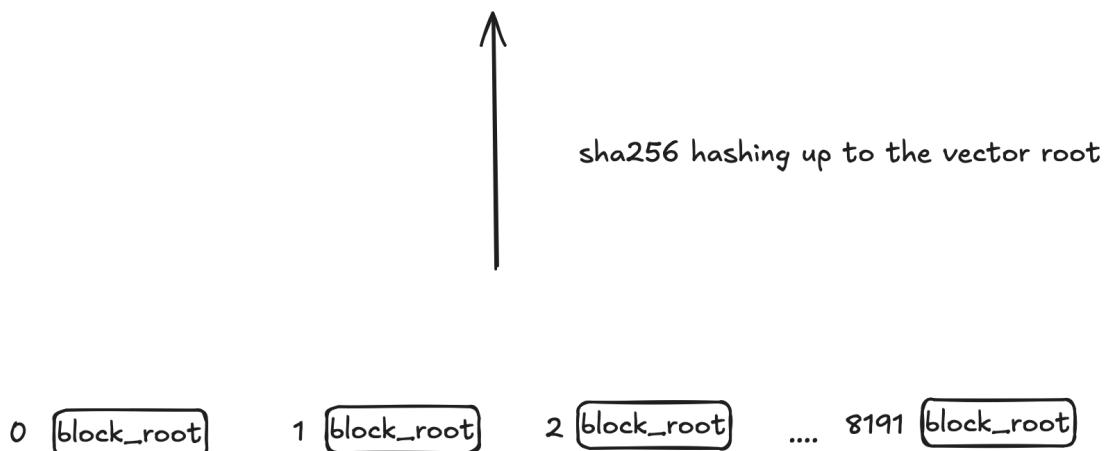
- depth = 6
- tail bits = 5 (5th index at the bottom of the `BeaconState` tree)



Now we have successfully crafted the path from `BeaconHeader` to `BeaconState` to the `block_roots` Vector. To understand the next step, we need to first dive into the merkle tree of the `block_roots` vector:

### `block_roots` properties:

- Vector type
- stores 8192 leaves at the bottom (ring buffer)
- No storage of length (unlike List)
- Each new slot, the latest block root will be written into `block_roots[slot % 8192]`
- Corresponds to “non-historical” (only 8192 leaves)
- depth = 13; no padding of leaves required due to exact leave size
- Each leave is the 32 bytes block\_root



The task is now to build the path from the vector root, down to the corresponding `block_roots` and then reconstruct the beacon block root with the corresponding witnesses.

This is achieved as follows:

`SSZ.intoVector(withdrawal.slot, 13)`

- `withdrawal.slot` represents the index of the slot within the Vector [non-historical; within 8192 slots of `proof.slot`]. This follows the same tail bits representation of left/right sections, as example, for index 8191, it is binary 111111111111; 13 right sections
- `depth = 13`

Since we have now achieved the path down to the corresponding `block_root` root, the next step is to get down from the `block_root` to the specific withdrawal. The `block_root` is the specific `BeaconBlockHeader` at the withdrawal slot. From there on, the next steps are:

- `BeaconBlockHeader (withdrawal slot) -> BeaconBlockBody`
  - `SSZ.from[4,3]`
- `BeaconBlockBody -> ExecutionPayload`
  - `SSZ.from[9,4]`
- `ExecutionPayload -> withdrawals [list]`
  - `SSZ.from[14,5]`
- Fetch the specific withdrawal via `_withdrawalNum`
  - `SSZ.intoVector(_withdrawalNum, 5)`
  - This should be `intoList(_withdrawalNum, 4)` but still works

With that, the path value successfully constructs the path down from `BeaconHeader` to `BeaconState` to the specific `block_roots`. This then allows to traverse the path using the provided `WithdrawalProof.withdrawal`, merkleising it and building up the path with the corresponding witnesses.

#### TLDR Path:

```
BeaconBlockHeader -> BeaconState  
BeaconState -> block_roots [Vector]  
block_roots toVector -> withdrawalSlot [BeaconBlockHeader]  
BeaconBlockHeader -> body_root [BeaconBlockBody]  
body_root -> ExecutionPayload  
ExecutionPayload -> withdrawals [List]  
withdrawals toList -> withdrawalNum
```

#### BeaconState to BlockRoot via historical proof

Within the historical proof scenario, roots for historical blocks are not stored within `block_roots` but instead within `historical_summaries`. The methodology to create the path (from the beginning) is:

```
BeaconBlockHeader -> BeaconState  
BeaconState -> historical_summaries [List]  
historical_summaries toList -> HistoricalSummary  
HistoricalSummary -> block_summary_root  
block_summary_root toVector -> withdrawalSlot [BeaconBlockheader]  
BeaconBlockHeader -> body_root [BeaconBlockBody]  
body_root -> ExecutionPayload  
ExecutionPayload -> withdrawals [List]  
withdrawals toList -> withdrawalNum [leaf]
```

All of this follows the already known logic within SSZ [from/intolist/intovector]. A small but important detail is the access of `historical_summaries` which only started from Capella and therefore requires offset deduction:

```
path = SSZ.concat[path, SSZ.intolist(uint256[_pastSlot] / slotsPerHistoricalRoot -  
historicalSummaryOffset, 24)]
```

whereas `historicalSummaryOffset = 758`; the number of how many 8192 batches were existing at the time of Capella fork.

#### Appendix: restoreMerkleRoot

The `restoreMerkleRoot` function is responsible for restoring the merkle root and the process is as follows:

- Sanity check to ensure witnesses length is sufficient
  - $> [2^{*[witnesses.length+1]}] > gindex$
- Execute as long as `gindex != 1` [root is not reached]
  - If `gindex = odd`, then `sha256[witness, value]`
  - If `gindex = even`, then `sha256[value, witness]`
  - Divide `gindex` by 2 to move up to the next depth and continue the same logic

Following this approach, the beacon block root will be reconstructed via the provided witnesses and the leaf [which was merkleised].

#### Appendix: gindex

The `gindex` is the global index for a leaf within the whole merkle tree for `BeaconBlockHeader`. It is important to note this distinct difference between leaves within subtrees. The higher the depth, the larger the `gindex`. Concatenating sequent paths and then putting the result into the `toIndex` function will always yield the `gindex` of the leaf which is either the `Withdrawal` data or the `Validator` data, merkleised.

The `gindex` can be simply reproduced by concatenating the different path decisions, starting from the root of the tree and going downwards will yield a binary sequence. This binary sequence can then be converted to a decimal number, which is exactly the `gindex`.

#### Core Invariants:

INV 1: `verifyValidator` and `verifyWithdrawal` must be idempotent

INV 2: [BeaconStateVerifier.verifyValidator](#) must ensure that a validator with correct credentials is existing on the CL

INV 3: `verifyWithdrawal` must ensure that a withdrawal has indeed happened

INV 4: Control-flow within `verifyWithdrawal` must differ between non-historic and historic withdrawals

#### Privileged Functions

- none

Issue_136	Insufficient validation allows for spoofing gindex
Severity	High
Description	<p>Rocket Pool's Megapool flow stakes validators via <code>RocketMegapoolManager.stake</code>.</p> <p>The manager relies on an external <code>BeaconStateVerifier</code> to validate a <code>ValidatorProof</code>: it checks the validator's <code>withdrawalCredentials</code> against the megapool's credentials and checks the <code>pubkey</code> against the megapool's stored <code>pubkey</code>.</p> <p>It then calls the <code>stake</code> function on the <code>MegapoolDelegate</code> contract, storing the <code>validatorIndex</code> provided in the proof as <code>validatorInfo.validatorIndex</code>.</p> <p>Subsequent lifecycle steps (<code>notifyExit</code>, <code>notifyNotExit</code>, <code>notifyFinalBalance</code>) verify proofs by checking equality of the proof's <code>validatorIndex</code> with the stored <code>validatorInfo.validatorIndex</code>.</p> <p>An integer overflow vulnerability exists in the <code>intoList()</code> function due to missing bounds check. This flaw allows a malicious actor to craft a very large, non-existent <code>validatorIndex</code> that, when processed by <code>toIndex()</code>, produces the same generalized index (<code>gindex</code>) as a different, valid <code>validatorIndex</code>.</p> <p>The overflowing bits from the oversized index when added to a list or vector can "bleed" into the position of the anchor bit during <code>gindex</code> calculation in the <code>toIndex</code> function. The normal structure of the <code>toList</code> function produce a path which is structured as follows:</p> <p>  - - bits 256 to 8--  &lt;empty&gt;&lt; INDEX &gt;  &lt;log2length + 1&gt;</p> <p>When converting the path to a generalized index this is done as follows:</p> <pre>uint256 pathLength = uint8[_path._data]; uint256 anchor = uint256[1] &lt;&lt; pathLength; return [_path._data &gt;&gt; 8] / anchor;</pre>

However, the critical invariant here is that the path cannot exceed the length which is determined by the placement of the anchor bit. Thus under normal usage, the structure of the gindex will look like this

*<anchor bit><path>eg. gindex in binary = 0x1001010*

the anchor is on the 6th bit [ie  $2^6$ ], whilst the path is 001010 [provides directions in the 6 steps to the required index needed].

If the path exceeds the length predetermined by the anchor bit, the right shift of 8 bits can align unexpectedly with the anchor, which will cause duplicate gindexes for different validator indexes.

Example:

```
validatorIndexA 1
validatorIndexB 1125899906842625 [aka  $2^{50}+1$ ]
SSZ.toIntList[validatorIndexA, 40] 297
SSZ.toIntList[validatorIndexB, 40] 288230376151712041
finalPathA 114278840544526642
finalPathB 402509216696238386
SSZ.length[finalPathA] 50
SSZ.length[finalPathB] 50
tolIndex[finalPathA] 1572301627719681
tolIndex[finalPathB] 1572301627719681
```

As can be seen in the above example

1125899906842625 == 1 as far as gindex is concerned.

This is a cross-module invariant violation: the protocol now believes the validator's beacon index is  $2^{50} + 1$ , while the canonical beacon index is 1. Since all later exit proofs compare the proof's index to the stored index, the attacker can pass `notifyExit` / `notifyNotExit` again by reusing the overflowed index [if the verifier allows the same alias], but cannot satisfy `notifyFinalBalance` because the withdrawal record comes with the true validator index [1] that won't equal  $2^{50} + 1$ . Finalization permanently fails.

	<p>This has the following concrete impacts:</p> <ul style="list-style-type: none"> <li>a) Permanent finalization lock: The validator's exit can never be finalized inside Rocket Pool. That blocks bond release and correct capital accounting.</li> <li>b) Monitoring blind spot: With <code>validatorInfo.validatorIndex</code> set to an out-of-range/nonexistent index [e.g., <math>2^{50} + 1</math>], normal off-chain watchers won't observe the real validator's exit/withdrawals, increasing the chance of late/never notification and unclear operational state.</li> </ul> <p>Important: There are multiple other data types which are not correctly sanitized, such as:</p> <ul style="list-style-type: none"> <li>- <code>withdrawalNum</code></li> <li>- <code>_pastSlot</code></li> <li>- <code>_index [SSZ]</code></li> </ul> <p>While we couldnt identify further impacts, we highly recommend sanitizing all parameters.</p>
<b>Recommendations</b>	Consider sanitizing all parameters to their correct data types.
<b>Comments / Resolution</b>	Resolved, stricter checks/types have been implemented.

<b>Issue_137</b>	Slightly ambiguous behavior during <code>notifyNotExit</code> [FOLLOWUP]
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>The <code>notifyNotExit</code> function checks that</p> $\_slotTimestamp \geq \text{validator.lockedTime}$ <p>while at the same time, <code>_slotTimestamp</code> returns the Parent Block Root (which was earlier). If now <code>lockedTime = 100</code> and we were not exiting at time = 88 (but we were at 100), we can now use <code>_slotTimestamp = 100</code> to satisfy the condition but it returns the older Block Root (at t=88)</p>
<b>Recommendations</b>	Consider acknowledging this issue.
<b>Comments / Resolution</b>	Acknowledged.

<b>Issue_138</b>	<code>BlockRoots</code> fetching returns a slot which is -1 from <code>_slotTimestamp</code> [FOLLOWUP]
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	<p>The <code>BeaconRoots</code> contract returns the parent <code>BlockRoot</code> from the <code>BlockRoot</code> of the timestamp we are providing, which is 12 seconds later.</p> <p>This is slightly falsifying the recency check as it is essentially one slot older.</p>
<b>Recommendations</b>	Consider acknowledging this issue.
<b>Comments / Resolution</b>	Acknowledged.

Issue_139	Incorrect usage of <code>intoVector</code> for list
Severity	Informational
Description	<p>Fetching the withdrawal from the withdrawals list within <code>ExecutionPayload</code> is using <code>intoVector</code>, while it actually is a list. Usually this does not work because of the distinct difference between List and Vector.</p> <p>However, due to the fact that there is an additional +1 for the extra depth, this works.</p>
Recommendations	Consider acknowledging this issue in an effort to prevent code changes.
Comments / Resolution	Resolved.

## BlockRoots

The [BlockRoots](#) contract is a simple handler contract which fetches the Beacon Root from the [BeaconRoots](#) contract on the EL. It acts as an intermediate contract between the [BeaconStateVerifier](#) and the [BeaconRoots](#) contract and is responsible for fetching the Beacon Root for a given slot. It does so by following EIP4788 and executing a staticcall with slot+1 to then return the Beacon Root at slot.

### Appendix: Beacon Block Root

Each execution block contains the parent's beacon block root. The [BeaconRoots](#) contract contains a buffer with 8191 roots which represent the historic state on the beacon chain. This root is then used as verification tool against the traversed route for the [WithdrawalProof](#) or [ValidatorProof](#).

### Core Invariants:

INV 1: `getBlockRoot` must advance `slotTimestamp` if return value is non-existent

### Privileged Functions

- none

Issue_140	Potential side-effects from proof.slot advancement
Severity	Informational
Description	<p>The <code>BlockRoots</code> helper (used by the proof verifier) retrieves a beacon block root by slot. It computes the timestamp for slot+1, enforces the history window, then walks forward by <code>secondsPerSlot</code> until it finds any root in the <code>BeaconRoots</code> contract.</p> <p>As a result, if the root for the exact <code>_slot</code> is missing (for whatever reason), <code>getBlockRoot[_slot]</code> can return the root of a later slot S'.</p> <p>The Rocket Pool proof flows (<code>stake</code>, <code>dissolve</code>, <code>notifyExit</code>, <code>notifyNotExit</code>, <code>notifyFinalBalance</code>) then rely on <code>BeaconStateVerifier.verify</code> to validate a <code>ValidatorProof/WithdrawalProof</code> against that root.</p> <p>There is no guarantee in <code>BlockRoots.getBlockRoot</code> that the returned root is for the same slot as requested; it can silently advance to the next available root. This can happen that the verification is effectively anchored to an unrelated later block.</p> <p>At slot S, a withdrawal may be present in the <code>ExecutionPayload.withdrawals</code> of that block.</p> <p>At a later slot S' (or at an epoch boundary), the same withdrawal may only be referenced in the historical summaries and no longer be in the live list of that block.</p> <p>Since applying the beacon state logic to megapools, future upgrades must maintain backwards compatibility with fork logic and withdrawal proofs. This exploit can be used to maliciously bypass fork choice when missed blocks occur before hard fork.</p>
Recommendations	<p>Consider acknowledging this issue. We have two comments:</p> <ul style="list-style-type: none"> <li>a) It seems to be a scenario that will never happen because it is expected that all roots are properly stored</li> <li>b) Even if the slotTimestamp (and the corresponding root)</li> </ul>

	advances, we could not find any exploitable state [other than potential reverts and re-execution of tx]
<b>Comments / Resolution</b>	Resolved. This contract has been removed in the resolution 1 commit.

## LinkedListStorage

The `LinkedListStorage` contract is a helper contract which is used by the `DepositPool` to manage the Megapools within the standard and express queue. It exposes three main functions:

- a) `enqueueItem`: This function is called whenever a new validator for a megapool is being created. It simply adds the validator item to the tail of the queue
- b) `dequeueItem`: This function is called whenever the head of the queue is assigned. It simply removes the item from the queue
- c) `removeItem`: This function is called whenever a validator decides to exit the queue before it is assigned, it will simply remove the item from the queue

The queue structure is a double ended queue with the head on the top and the tail on the bottom. The head is used to determine which index is assigned next and every item in the list has a predecessor (`prev`) and successor (`next`), unless when the item is at the tail (no successor) or head (no predecessor). It is furthermore ensured that during an item removal, successor and predecessor are adjusted to ensure the integrity of the list.

### Appendix: data

The `data` variable is unique for the standard queue and for the express queue and packs the following values into a `uint256`:

- start pointer [192 - 255]
- end pointer [128 - 191]
- length of the queue [64 - 127]
- unused [0 - 63]

### Appendix: item

The `item` variable is unique for the standard queue and for the express queue and packs the following values into a `uint256`:

- receiver [96 - 255]
- validatorId [64 - 95]
- suppliedValue [32 - 63]
- requestedValue [0 - 31]

## Core Invariants:

INV 1: `LinkedListStorage.peekItem` must return the head of the queue

INV 2: `item.receiver` must always point to a Megapool

INV 3: `data.end` must move every time `enqueueItem` is called

INV 4: item at the start must have successor [next] but no predecessor

INV 5: item at the end must predecessor [prev] but no successor

INV 6: items that are not start nor end must have predecessor [prev] and successor [next]

INV 7: `start` must always point to the oldest remaining item in the queue that has not yet been processed

INV 8: During `dequeueItem`, if a deque'd item has a successor, its predecessor must be removed

INV 9: During `removeItem`, if a removed item has a successor, its predecessor must be removed

INV 10: Any item which is considered during `_dequeueItem` should never have a predecessor

INV 11: After `_dequeueItem` is called with the only item in the list, `start` and `end` must be set to zero

INV 12: After `_removeItem` is called with the only item in the list, `start` and `end` must be set to zero

## Privileged Functions

- none

Issue_141	Lack of state clearance will result in broken Megapool assignment
Severity	High
Description	<p>LinkedListStorage maintains an address-keyed, doubly linked list with explicit start/end pointers and per-node prev/next indices, stored in RocketStorage.</p> <p>The public read functions [<code>getHeadIndex</code>, <code>getItem</code>, <code>getIndexOf</code>, <code>scan</code>, <code>peekItem</code>] trusts two invariants: [1] start points to the current head index, and [2] the per-index prev/next links are consistent with membership.</p> <p>The write functions [<code>enqueueItem</code>, <code>dequeueItem</code>, <code>removeItem</code>] are responsible for preserving those invariants.</p> <p>There are multiple places where the write functions leaves dangling links and stale slot contents, which can later be reinterpreted as live:</p> <p><b>First enqueue on an empty queue does not clear links for the new head:</b></p> <p>In <code>_enqueueItem</code>, when <code>endIndex == 0</code>, the code sets <code>start = newIndex</code> and <code>end = newIndex</code> but does not write <code>prev[newIndex] = 0</code> and <code>next[newIndex] = 0</code>.</p> <p>If <code>newIndex</code> reused a slot from a prior queue, its <code>.next[newIndex]</code> may still point at an old successor. The next <code>dequeue</code> will read this stale successor and set <code>start</code> to an incorrect index.</p> <p><b>_dequeueItem does not clear the dequeued node's links:</b></p> <p>The function advances <code>start</code> to <code>next[start]</code> and clears <code>prev[next]</code> (or <code>end</code>), but it never sets <code>next[oldStart] = 0</code> or <code>prev[oldStart] = 0</code>. This leaves an orphaned next link that can be followed later if the slot is reused as a head again (see #1).</p> <p><b>Removed slots keep their packed item forever:</b></p>

Neither `_dequeueItem` nor `_removeItem` clears `.item[index]`. As long as start/iteration can be pushed onto a stale index, `peekItem` will unpack and return a removed element as if it were at the head [because `peekItem` simply reads `.item[start]`].

**Head detection in `removeItem` depends on possibly stale `prevIndex`:**

The branch if `[prevIndex > 0] { ... }` else `{ /* first item */ }` assumes `prev[index]` truthfully encodes head membership. With #1/#2, `prev[index]` can be stale, so head removal may run the wrong branch, producing wrong start/end.

Therefore, `getHeadIndex`, `peekItem`, and `scan` can return wrong head indices and stale items.

Below we will demonstrate the impact where `peekItem` returns an incorrect value and as result fully breaks the `DepositPool._assignMegapools` control-flow:

**item A will be enqueue'd at index = 1**

- `endIndex = 0`
- `newIndex = 1`
- entering else-clause; not setting prev and next
- `start = 1`
- `end = 1`
- `item[1] = A`
- `index[A] = 1`
- `length = 1`

**item B will be enqueue'd at index = 2**

- `endIndex = 1`
- `newIndex = 2`
- entering if-clause, set prev and next
  - `next[1] = 2`
  - `prev[2] = 1`
- `start = 1`
- `end = 2`

- item[2] = B
- index[B] = 2
- item[1] = A
- index[A] = 1
- length = 2

**dequeue item A**

- start = 1
- nextItem = 2
- start = 2
- end = 2
- index[A] = 0
- index[B] = 2
- prev[2] = 0
- next[1] = 2
- item[2] = B
- item[1] = A
- length = 1

**dequeue item B**

- start = 2
- nextItem = 0
- start = 0
- index[B] = 0
- index[A] = 0
- prev[2] = 0
- next[1] = 2
- item[2] = B
- item[1] = A
- end = 0
- length = 0

**item C will be enqueued at index = 1**

- endIndex = 0
- newIndex = 1
- start = 1
- item[1] = C
- item[2] = B
- index[C] = 1
- end = 1
- length = 1

	<ul style="list-style-type: none"><li>- prev[2] = 0</li><li>- next[1] = 2</li></ul> <p><b>dequeue item C</b></p> <ul style="list-style-type: none"><li>- start = 2</li><li>- end = 1</li><li>- item[1] = C</li><li>- item[2] = B</li><li>- index[C] = 0</li><li>- prev[2] = 0</li><li>- next[1] = 2</li><li>- length = 0</li></ul> <p><b>Enqueue E</b></p> <ul style="list-style-type: none"><li>- endIndex = 1</li><li>- newIndex = 2</li><li>- next[1] = 2</li><li>- prev[2] = 1</li><li>- item[2] = E</li><li>- index[E] = 2</li><li>- start = 2</li><li>- end = 2</li><li>- length = 1</li></ul> <p><b>Enqueue F</b></p> <ul style="list-style-type: none"><li>- endIndex = 2</li><li>- newIndex = 3</li><li>- next[2] = 3</li><li>- prev[3] = 2</li><li>- next[1] = 2</li><li>- prev[2] = 1</li><li>- item[3] = F</li><li>- index[F] = 3</li><li>- index[E] = 2</li><li>- item[2] = E</li><li>- start = 2</li><li>- end = 3</li></ul> <p><b>Remove item E</b></p> <ul style="list-style-type: none"><li>- prevIndex = 1</li><li>- nextIndex = 3</li><li>- next[1] = 3</li></ul>
--	--

	<ul style="list-style-type: none"> <li>- prev[3] = 1</li> <li>- index[E] = 0</li> <li>- next[2] = 0</li> <li>- prev[2] = 0</li> <li>- item[3] = F</li> <li>- index[F] = 3</li> <li>- item[2] = E</li> <li>- start = 2</li> <li>- end = 3</li> <li>- length = 1</li> <li>- queue has live element = 3 but points as start to 2</li> </ul> <p>peekItem will return 2 and uses the stale item[2] = E which was just removed</p> <p><b>Another iteration of actions would be:</b></p> <ul style="list-style-type: none"> <li>- Enqueue A</li> <li>- Enqueue B</li> <li>- Enqueue C</li> <li>- Remove B</li> <li>- Dequeue A</li> <li>- Dequeue C</li> <li>- Enqueue D</li> <li>- Dequeue C</li> <li>- Enqueue E</li> </ul> <p>Result is similar as above start = 3 end = 2 and E points to 2 while now C will be used as head</p>
<b>Recommendations</b>	Consider correctly clearing all states.
<b>Comments / Resolution</b>	Resolved, all states are now cleared.

Issue_142	Redundant SSTORE during head removal with successor
Severity	Informational
Description	<p>While handling head removal (<code>prevIndex == 0</code> and <code>nextIndex &gt; 0</code>), <code>_removeItem</code> writes the successor's prev pointer twice:</p> <p>In the head branch, it sets <code>start = nextIndex</code> and immediately writes <code>prev[nextIndex] = 0</code>.</p> <p>Immediately after, the generic "if [<code>nextIndex &gt; 0</code>]" branch writes <code>prev[nextIndex] = prevIndex</code> again, which is also 0 in the head case.</p> <p>This is a redundant SSTORE of the same value to the same slot.</p>
Recommendations	Consider acknowledging this issue. There is no impact besides a slight gas inefficiency and potential changes may result in side-effects.
Comments / Resolution	Acknowledged.

## SSZ

The SSZ contract is a helper contract which is used by the [BeaconStateVerifier](#). It exposes multiple different functions for crafting the path and recomputing the root from the crafted path and the provided leaf and witnesses. All important logic is described within the Appendices of the [BeaconStateVerifier](#).

### Core Invariants:

INV 1: `toList` must add an extra 1 for the length

INV 2: `restoreMerkleRoot` must enforce the correct length of witnesses

### Privileged Functions

- none

Please see [BeaconStateVerifier](#) for input validation recommendations.

Issue_143	0 gindex returns the leaf passed successfully
Severity	Informational
Description	This issue is non exploitable since gindex of zero is not used directly. However, if an accident occurs that somehow lowers the gindex unexpectedly, say some unknown issue during concat, then <code>restoreMerkleRoot()</code> will return the leaf directly without hashing. This may break expected logic of merkle root verification.
Recommendations	Recommend reverting for <code>gindex == 0</code>
Comments / Resolution	Acknowledged.

<b>Issue_144</b>	Reversed NATSPEC for intoVector/intoList
<b>Severity</b>	<b>Informational</b>
<b>Description</b>	The NATSPEC for both functions is reversed. This will confuse third-party reviewers.
<b>Recommendations</b>	Consider correcting the NATSPEC.
<b>Comments / Resolution</b>	Resolved.

## Base

### RocketBase

The RocketBase contract is the backbone contract of the system which includes all base functionalities such as fetching, setting and deleting storage variables. It is inherited by almost all contracts in the system and is essential for the protocol to stay synced with the external RocketStorage contract.

This contract has been changed for the Saturn upgrade.

### Appendix: Storage Interaction

The RocketStorage contract is externally deployed and keeps track of all storage variables for the overall architecture. It exposes externally callable functions which are solely called by the RocketBase contract, which makes the RocketBase contract the only contract that is capable of interacting with the RocketStorage.

### Privileged Functions

- none

No issues found.

## RocketStorage

The [RocketStorage](#) contract is an externally deployed contract which holds the storage for the RocketPool architecture. It implements strict access control that only network contracts can change the storage. Furthermore, it incorporates logic for setting the guardian and the withdrawal address for nodes.

**This contract has not been changed for the Saturn upgrade.**

### Appendix: Storage Layout

The storage layout is uniform within the whole Rocket Pool architecture and centralized in a single [RocketStorage](#) contract, which acts as a global key–value store shared across all modules.

Every piece of protocol state is stored under a keccak256-derived key.

Different types of data use different mappings:

- mapping[bytes32 => uint256] uintStorage;
- mapping[bytes32 => address] addressStorage;
- mapping[bytes32 => bool] booleanStorage

...and so on for string, bytes, int, bytes32.

### Core Invariants:

INV 1: If a node has no withdrawalAddress set; own address will be returned

INV 2: Only the withdrawal address of a node or the node [in case no withdrawal address is set] can change the withdrawal address

INV 3: guardian change must go through a 2-step process

INV 4: withdrawalAddress change can go optionally through a 2-step process

## Privileged Functions

- setAddress
- setUint
- setString
- setBytes
- setBool
- setInt
- setBytes32
- deleteAddress
- deleteUint
- deleteString
- deleteBytes
- deleteBool
- deleteInt
- deleteBytes32
- addUint
- subUint

No issues found.

## RocketVault

The [RocketVault](#) contract is a simple custodian contract for native ETH and every ERC-20 token. It is however only used with the RPL token. It stores both tokens on behalf of various contracts.

**This contract has not been changed for the Saturn upgrade.**

### Appendix: ETH Deposits

Native ETH deposits are executed by the following contracts:

- DepositPool
- NodeDeposit
- NodeManager
- ClaimDAO
- MerkleDistributor
- RewardsPool

### Appendix: RPL Deposits

RPL Deposits are executed by the following contracts:

- DepositPool
- NodeDeposit
- NodeManager
- MerkleDistributor
- RewardsPool

### Privileged Functions

- depositEther
- withdrawEther
- withdrawToken
- transferToken
- burnToken

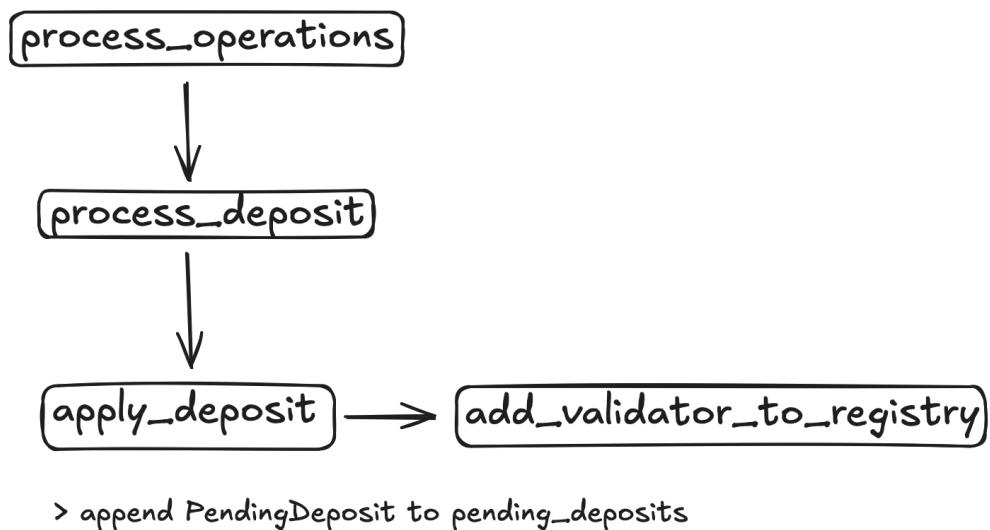
No issues found.

## Beacon Chain Appendices

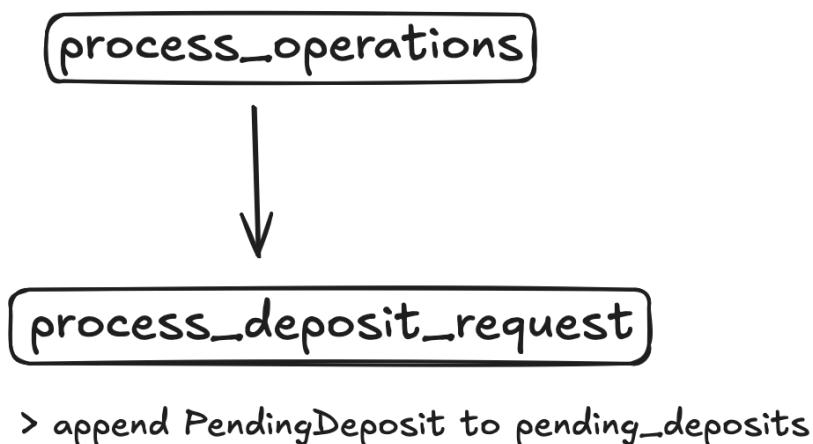
### Appendix Beacon Chain: Deposits

The deposit flow starts with the creation of a PendingDeposit container which is appended to the `pending_deposits` variable:

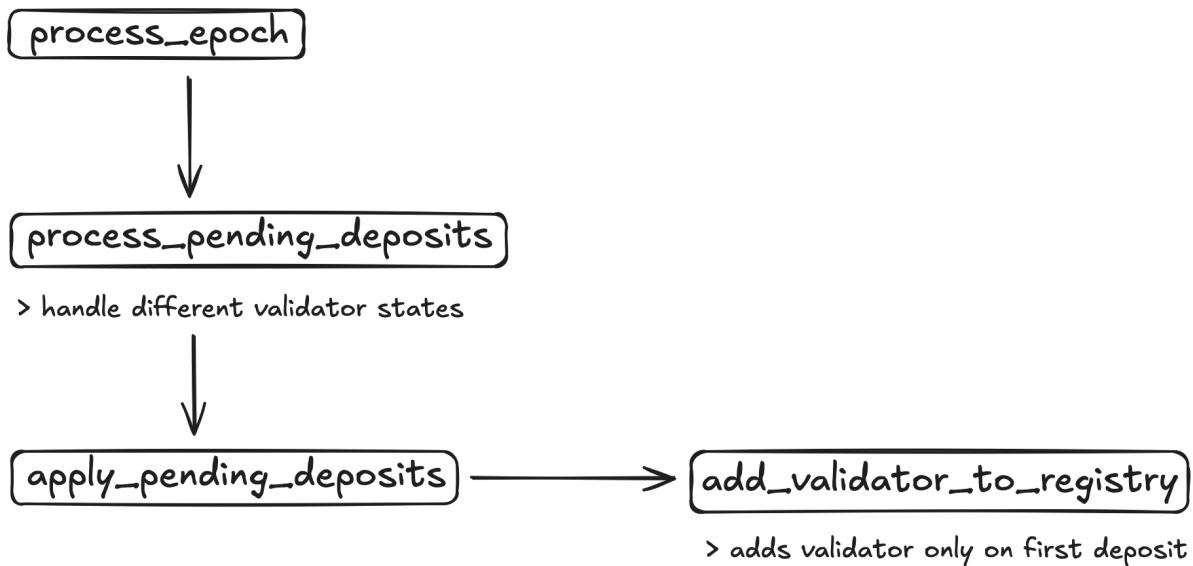
Pre-Electra Flow:



Post-Electra Flow:

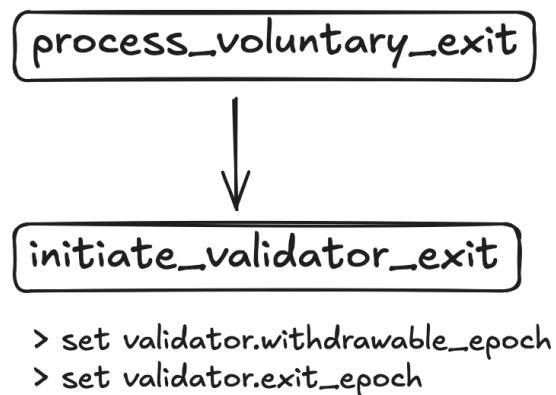


Within the next execution path, PendingDeposit containers within pending\_deposits are executed as follows:



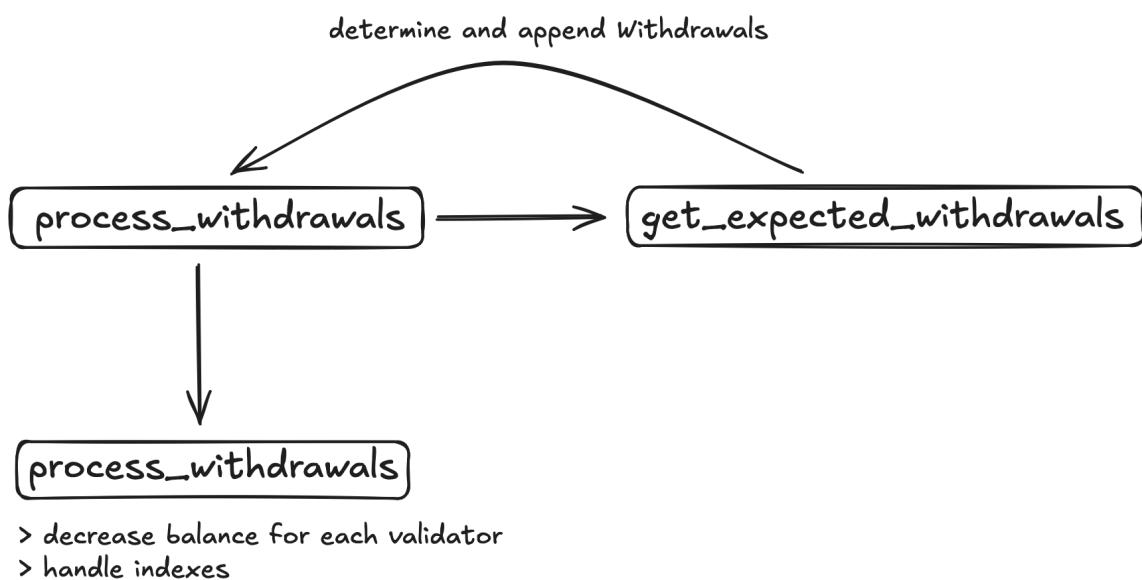
#### Appendix Beacon Chain: Withdrawals

Common withdrawals are starting with a withdrawal initiation using a validators BLS key. The flow for this is as follows:



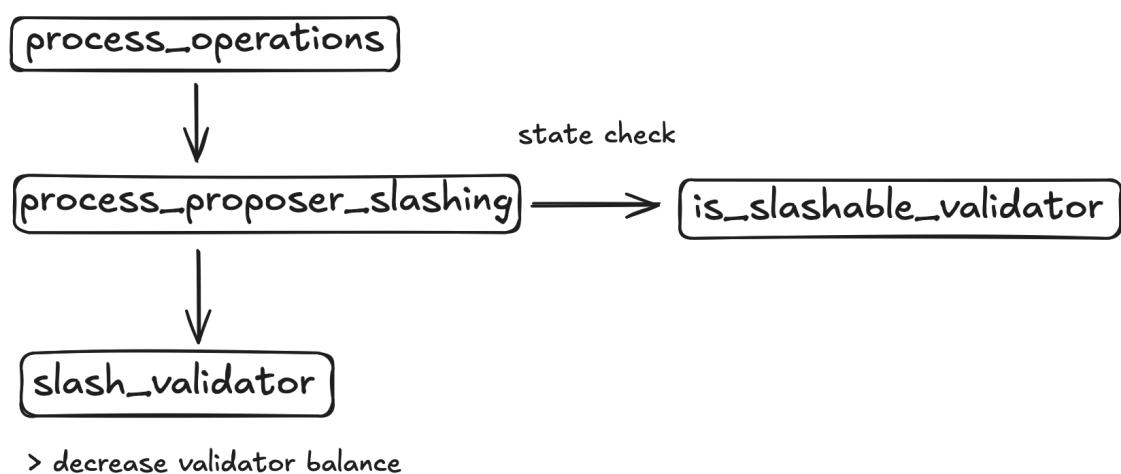
There is also the path of the execution-layer request. However, this is not relevant in the context of Rocketpool as this requires the withdrawal address [Minipool/Megapool] to call the system address (<https://eips.ethereum.org/EIPS/eip-7002>).

On each `process_block` execution, a scan through all validators (with certain boundaries) is executed and all valid withdrawals are then determined and executed:



## Appendix Beacon Chain: Slashing

Slashing happens under different circumstances and penalizes a validator as well as triggers an exit, the flow is as follows:



In the context of RocketPool, it is important to understand that slashing will initiate an exit which then requires the validator to be exited in the RocketPool accounting state as well. Furthermore, if slashing for an already exiting validator happens, it will extend the `withdrawable_epoch` 36.4 days into the future. Slashing is only allowed once.

## Appendix: Beacon Chain: 0x01 vs 0x02

There are two types of validators, whereas only the 0x01 validator is important in the RocketPool context.

### 0x01 – Execution-layer (ETH1) withdrawal credentials

#### Format:

- *0x01 / 11 zero bytes / 20-byte Ethereum address*

#### Meaning:

- Funds are withdrawn directly to the specified Ethereum address on the Execution Layer (EL).
- All full withdrawals [when a validator has exited and `withdrawable_epoch` is reached] go directly to this address.
- All partial withdrawals [excess balance above max effective balance] are also swept to this address automatically.

#### Properties:

- One-way: you cannot revert a validator back to BLS [0x00] or compounding [0x02].
- The validator is a “plain ETH-to-wallet” validator.
- Recommended for solo stakers, custodians, and protocols that want auto-payouts into an EOA or smart contract.

### 0x02 – Compounding (execution) withdrawal credentials

#### Format:

- *0x02 / 11 zero bytes / 20-byte Ethereum address*

#### Meaning:

- Funds are withdrawn to the given Ethereum address, but with compounding semantics.
- Instead of sweeping excess balance out as partial withdrawals, excess ETH above the max effective balance (32+ by default, up to the new

`MAX_EFFECTIVE_BALANCE_ELECTRA`] is re-deposited automatically to increase the validator's effective balance, subject to churn limits.

- This means the validator can grow above 32 ETH and accumulate larger effective balances.

#### Properties:

- Enables validators to automatically reinvest excess ETH into their balance, without operator interaction.
- Introduced with Electra to support larger validator sizes and reduce the operational overhead of splitting validators.
- Compatible with Execution Layer smart contracts [destination can be a contract], but semantics are different than 0x01.