sigma prime

Rocket Pool

Rocket Pool Protocol Review

Version: 2.0

May, 2021

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Rocket Pool smart contracts, smart node and developer library. The review focused solely on the security aspects of the Solidity and Golang implementation of the network, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality for the Rocket Pool smart contracts, smart node and developer library contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite and Round Two Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Rocket Pool smart contracts, smart node and developer library.

Overview

Rocket Pool is a decentralised staking network built to be compatible with the Ethereum 2.0 beacon chain (eth2). This staking network aims to allow users who don't possess the minimum ETH requirement (32) to become a validator or the necessary technical skills to run a node, to participate in staking and earn rewards.

Rocket Pool uses three distinct tokens:

- **RPL:** Native Rocket Pool token, used for DAO governance as collateral by protocol node operators and "oracle" nodes.
- **rETH:** Wrapper token for Ether staked using Rocket Pool, received by users in exchange for depositing ETH to Rocket Pool.
- **nETH:** Wrapper token for Ether that represents the node operator's share of an exited eth2 validator's balance, which can be later exchanged one-to-one for Ether once eth2 staking withdrawals are implemented. To be deprecated when eth2 staking withdrawals are implemented.

This staking service allows single stakers to deposit 16 ETH, with the protocol assigning 16 ETH from users depositing ETH and receiving rETH in return. This allows these single stakers to earn a commission rate on the

16 ETH assigned by the protocol, allowing node operators to effectively earn rewards for staking other users' ETH.

The codebase targeted in the second round of assessment had removed the nETH token in favour of reduced contract complexity and ensuring the system is better designed to work with ETH2 withdrawals when they are enabled.

Security Assessment Summary

This review initially targeted the following commits:

- `rocket-pool/rocketpool` : Smart contracts powering the Rocket Pool protocol.
  - Commit 026cc6c
- `rocket-pool/rocketpool-go` : Golang developer library for interacting with the Rocket Pool protocol.
  - Commit fb20418
- `rocket-pool/smartnode` : Rocket Pool node implementation.
  - Commit 0636339

The retesting review was conducted on the following commits:

- `rocket-pool/rocketpool` : Smart contracts powering the Rocket Pool protocol.
  - Commit 5fc97999
  - PR #201
- `rocket-pool/rocketpool-go` : Golang developer library for interacting with the Rocket Pool protocol.
  - Commit a8a504f
- `rocket-pool/smartnode` : Rocket Pool node implementation.
  - Commit 774ec28
- `rocket-pool/smartnode-install` : Rocket Pool node installation.
  - Commit d80f424

Note: the OpenZeppelin libraries and dependencies used in `rocketpool` *were excluded from the scope of this assessment.*

The manual code review section of the report, focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts. Specifically, their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focused on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [1, 2].

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

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`

Fuzzing activities leveraging go-fuzz have been performed by the testing team in order to identify panics within the code in scope. `go-fuzz` is a coverage-guided tool which explores different code paths by mutating input to reach as many code paths possible. The aim is to find memory leaks, overflows, index out of bounds or any other panics.

Specifically, the testing team produced the following fuzzing targets, shared with the development team:

- `FuzzByteArrayUnmarshalJSON`
- `FuzzDecodeAbi`
- `FuzzEncodeAbiStr`
- `FuzzMinipoolDepositString`
- `FuzzMinipoolDepositUnmarshalJSON`
- `FuzzMinipoolStatusUnmarshalJSON`
- `FuzzProposalStateString`
- `FuzzProposalStateUnmarshalJSON`
- `FuzzRocketPoolConfig`
- `FuzzUintegerUnmarshalJSON`
- `FuzzValidatorPubkeyUnmarshalJSON`
- `FuzzValidatorSignatureUnmarshalJSON`

These fuzzing targets have all been shared with the development as a by-product of this security review. Execution and instrumentation can be done using a script provided by the testing team (`fuzz_gen.sh`).

Additionally, the team produced a suite of python-based tests. These tests verify some of the core business logic as well as demonstrate some of the listed vulnerabilities and issues raised in this report.

The output of these tests are provided in the Appendix (Test Suite and Round Two Test Suite), and the implementations have been provided to the development team alongside this report.

Findings Summary

The testing team identified a total of thirty-three (33) issues during the first round of this assessment (`RP-1` to `RP-33`), of which:

- One (1) is classified as high risk,
- One (1) is classified as medium risk,
- Fourteen (14) are classified as low risk,
- Seventeen (17) are classified as informational.

The testing team identified a total of eight (8) issues during the second round of this assessment (`RP-34` to `RP-41`), of which:

- One (1) is classified as medium risk,
- Three (3) are classified as low risk,
- Four (4) are classified as informational.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Rocket Pool platform. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the code base, including comments not directly related to the security posture of Rocket Pool (e.g gas optimisations), are also described in this section and are labelled as *"informational"*.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
RP-01	Unprotected Storage Allows Compromise During Deployment	High	Resolved
RP-02	Insufficient Delay for `RocketNodeStaking.withdrawRPL()`	Medium	Resolved
RP-03	Node Operators Can Lose All Funds Via An Invalid Signature	Low	Closed
RP-04	Trusted DAO Member Multiple Vote via Replace	Low	Resolved
RP-05	Less Secure Node Account Can Set Withdrawal Address	Low	Resolved
RP-06	Delayed Refunds for "FullDeposit" Minipools	Low	Closed
RP-07	Unmarshalling Pointers as `nil`	Low	Open
RP-08	SSH Passphrase as Command Line Argument	Low	Resolved
RP-09	Insufficient Password Strength & Complexity Requirements	Low	Resolved
RP-10	Panic in the Unmarshalling of `ValidatorPublicKey` & `ValidatorSignature`	Low	Resolved
RP-11	Incorrect Access Control List for `_upgradeContract()` Function	Low	Resolved
RP-12	Network Contracts Have Unrestricted Access to Storage	Low	Closed
RP-13	Divide before Multiply	Low	Resolved
RP-14	Refund of Successful Challenge	Low	Closed
RP-15	Inaccurate calculation of `getTotalEffectiveRPLStake`	Low	Closed
RP-16	Reliance on ETH1 Provider	Low	Closed
RP-17	Ineffective RPL Staking Collateral	Informational	Open
RP-18	Likely Gas Savings When Setting RocketStorage Values	Informational	Resolved
RP-19	Gas Savings via Bulk and Update Storage Functionality	Informational	Open
RP-20	`RocketTokenRPL.swapTokens` gas savings	Informational	Resolved
RP-21	Unhandled Errors	Informational	Closed
RP-22	Unused and Lack of Constant Variables	Informational	Open
RP-23	Consolidation of `RocketDAONodeTrustedActions`	Informational	Resolved
RP-24	Lack of Input Validation	Informational	Open
RP-25	Suboptimal Definition of MiniPool Storage Layout	Informational	Resolved

RP-26	DAO Settings Checks	Informational	Open
RP-27	Rounding of Auction Bids	Informational	Open
RP-28	Improper Emitting of Events	Informational	Resolved
RP-29	Potential Settings “Getter” Gas Optimizations	Informational	Open
RP-30	RocketMinipool Deployment Gas Optimisations	Informational	Open
RP-31	Distributed Storage Key Namespace Design and Organisation	Informational	Open
RP-32	Functions Can Be Declared External For Gas Savings	Informational	Open
RP-33	Miscellaneous Rocket Pool Contract Issues	Informational	Open
RP-34	Inaccurate RPL Inflation When Minting Multiple Intervals At Once	Medium	Open
RP-35	Unexpected Behaviour If RPL Inflation Rate Set to Zero	Low	Open
RP-36	Node Operator Can Refuse to Distribute Minipool Funds	Low	Open
RP-37	Node Operator Can Revert `processWithdrawal()`	Low	Open
RP-38	Unhandled Errors — Round Two	Informational	Open
RP-39	Frequent RPL Reward Claim Requirement Unevenly Impacts Small Node Operators	Informational	Open
RP-40	Inconsistent User Deposit Gas Estimation	Informational	Open
RP-41	Miscellaneous Rocket Pool Contract Issues — Round Two	Informational	Open

RP-01	Unprotected Storage Allows Compromise During Deployment		
Asset	`rocketpool: RocketStorage.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The `onlyLatestRocketNetworkContract()` modifier allows direct modification of the RocketStorage contents during deployment. *Any* account can perform arbitrary storage modifications during the course of deployment, causing significant impact.

The current deployment method[1] involves multiple transactions and so is not *atomic*. Between the `RocketStorage` contract's initial deployment and when the boolean value `"contract.storage.initialised"` is set to `true` upon completion of the initial setup, any account can successfully modify any storage value.

Possible attacks and impacts include:

- Setting `"contract.storage.initialised"` before deployment has finished, making completed deployment impossible with that `RocketStorage` instance.
- Setting the address of `"rocketVault"` to a contract controlled by the attacker that allows them to withdraw any deposited funds.
- Setting `"contract.exists"` to an attacker controlled contract. This effectively introduces a *backdoor*, allowing an attacker to make arbitrary storage modifications at a later date.

As of testing, deployment involves about 223 individual transactions, so there is plenty of time for an attacker to interfere, even without front-running with a high gas price. Given that Rocket Pool is a high profile project and has some competitors, there is feasible motivation for a targeted attacker to exist. It would be complicated to effectively hide the `RocketStorage` address during deployment, as a targeted attacker could monitor the mempool and recent blocks for the relevant bytecode.

Although the attacks listed above are technically possible, any interference during deployment is straightforward to detect, so any catastrophic impacts (resulting in the loss of user funds) should not occur. A failed deployment would simply never be advertised as the canonical Rocket Pool instance.

In reality, these attacks could hinder a successful deployment (by repeatedly sabotaging deployment attempts) and cost the development team excessive Ether in gas costs.

Recommendations

Some DApps can take advantage of an atomic deployment by utilising a *"factory"* contract to bundle the deployment and setup of multiple contracts into a single transaction. This ensures no other transaction can affect a partially deployed DApp. However, due to the large number of contracts that must be deployed during setup, this method is not feasible for Rocket Pool.

[1] A TruffleSuite migration defined at `rocketpool/migrations/2_deploy_contracts.js`

State changing operations in `RocketStorage` are protected by the `onlyLatestRocketNetworkContract()` modifier, which consists of the following check:

```
if (boolStorage[keccak256(abi.encodePacked("contract.storage.initialised"))] == true) {
    // Make sure the access is permitted to only contracts in our Dapp
    require(boolStorage[keccak256(abi.encodePacked("contract.exists", msg.sender))], "Invalid
         or outdated network contract");
}
```

The testing team recommend changing this modifier to contain a check equivalent to the following:

```
if (boolStorage[keccak256(abi.encodePacked("contract.storage.initialised"))] == true) {
    // Make sure the access is permitted to only contracts in our Dapp
    require(boolStorage[keccak256(abi.encodePacked("contract.exists", msg.sender))], "Invalid
         or outdated network contract");
} else {
    // Only Dapp and the guardian account are allowed access during initialisation.
    require((
        boolStorage[keccak256(abi.encodePacked("contract.exists", msg.sender))]
        || boolStorage[keccak256(abi.encodePacked("access.role", "guardian", msg.sender))]
    ), "Invalid or outdated network contract");
}
```

This would allow the deployer account direct access to storage during initialisation while appropriately preventing interference by an external attacker. The additional gas costs incurred by adding such a check would only be experienced during deployment.

In order to allow deployed contracts to make storage changes in their constructor, the existing migration script would need to be modified to register the contract with `RocketStorage` prior to deployment. This would involve pre-calculating the address of the new contract from an appropriate nonce value, or deploying via a factory contract using the `CREATE2` opcode. This pre-calculation is somewhat complicated but achievable (provided the deployer account is not used for unrelated transactions during the course of deployment).

A simpler alternative could be to instead authenticate the `"guardian"` via `tx.origin`, but this has its own safety concerns. This would allow contracts deployed by the `"guardian"` to modify storage without needing to be pre-registered, but would be unsafe should the deployer account interact with any external contracts during the deployment.

Resolution

This was resolved in commit 495a51f that requires, during deployment, transactions be sent from registered network contracts or originate from the `"guardian"` account (via `tx.origin`).

RP-02	Insufficient Delay for `RocketNodeStaking.withdrawRPL()`
Asset	`rocketpool: RocketNodeStaking.sol`
Status	**Resolved:** See Resolution
Rating	Severity: Medium Impact: Medium Likelihood: Medium

Description

At present, `withdrawRPL()` only checks the cooldown based on the last block in which the node had increased their stake.

The require on line [165] checks that a node hasn't staked recently:

```
require(block.number.sub(getNodeRPLStakedBlock(msg.sender)) >=
        rocketDAOProtocolSettingsRewards.getRewardsClaimIntervalBlocks(),
  "The withdrawal cooldown period has not passed");
```

Due to the inherent reporting delay, time to achieve an appropriate quorum, and the fact that the Trusted Node DAO reports only finalized states, it is highly likely that a node operator will know that their staked RPL is eligible for slashing before this is reported to `RocketMiniPool`.

As the current restriction will be satisfied as long as a node has not increased their stake within the recent period, a node is incentivised to withdraw all but the minimum stake. Given that a node will then only be slashed the minimum amount, there is no reason for the protocol to reward staking more than the minimum RPL (i.e. any amount over the minimum is unlikely to be used to reimburse rETH balance losses and will not be used as collateral).

Recommendations

The testing team recommends the introduction of a fixed delay between when a node first signals a desire to withdraw staked RPL and when such a withdrawal is permitted, such that this delay is larger than (at minimum) the average time taken to report an RPL–slashable event.

A potential implementation could involve requiring that a node wishing to withdraw should avoid claiming RPL rewards for an entire claim period before withdrawal is permitted. While this results in some loss in RPL rewards, it avoids the need for a separate “signal withdraw” transaction (and associated gas costs).

Even with this increased delay, it is likely that a node will know that it is eligible for RPL slashing long before it has exited and is reported as `Withdrawable`. Although this doesn't resolve the situation entirely, consider introducing another status `Exited`, that is set by the Trusted DAO Nodes when the validator is no longer active but before it is `Withdrawable`. For slashed validators, this should be set as soon as the slashing has been included in a finalized block (i.e. when it is “active slashed”, not waiting for any later penalties to be applied). While `Exited`, prevent the node from withdrawing any staked RPL.

Resolution

This was resolved in commit 40321a8 that only allows node operators to withdraw staked RPL in excess of the maximum amount eligible for rewards (by default at a 150% collateralisation ratio). Because node operators cannot withdraw their staked RPL until the validator is `Withdrawable`, the node operator cannot avoid any deserved slashing penalties.

RP-03	Node Operators Can Lose All Funds Via An Invalid Signature		
Asset	`rocketpool: RocketNodeManager.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

An Eth2 deposit can be submitted with an invalid signature, resulting in lock/loss of all funds such that only the 10% RPL stake can be relied upon to recover funds for the network.

Normally, any balance losses are recovered from the node operator's ether collateral, protecting rETH holders. Additionally, with the current Eth2 penalty calculations, it is highly unlikely that any balance losses due to slashing or inactivity could exceed 16 ETH. If even possible, the slashing would need to occur as part of a widespread slashing event (in which many colluding validators are slashed), or the validator would need to be inactive for a *very* long time during an extended period of non-finality.

However, invalid deposits can be submitted to the Eth2 deposit contract (which doesn't verify signatures, only deposit merkle root) such that, when processed by the Eth2 beacon chain, the deposit is discarded and the balance is never associated with a validator. With the entire balance lost, only the staked RPL (a minimum of $10\% = 1.6$ ETH) can be used to recover funds. This is the only identified, feasible way a node operator can cause the exchangeable value of rETH to decrease.

As an invalid deposit will result in the loss of the entirety of the node operator's 16 ETH + RPL collateral, this is costly to exploit at scale. It may, however, be possible to amplify a smaller-scale exploit to create and profit from a reputational scare (e.g. by shorting RPL).

This is also a significant impact available to trusted nodes with unbonded validators. Normally, a significant portion of trusted nodes would need to maliciously collude in order to negatively affect Rocket Pool. With unbonded validators, a malicious node can lock a relatively significant amount of user ETH (though this is somewhat offset by the Trusted Node RPL bond requirement).

It may be possible that, when withdrawals are implemented, the Eth2 system withdrawal contract allows recover of invalid deposits back to their withdrawal address. However, this cannot be relied upon.

Recommendations

As BLS signatures cannot currently be verified on-chain, this may be a risk that cannot be feasibly removed until BLS precompiles are available for EVM contracts (e.g. via EIP 2537). While it could be possible to use a fixed withdrawal address and have the node provide precomputed signature pairs that must be verified before they are allowed to create a minipool, this is not very feasible given the decentralised design of Rocket Pool (you would need to rely on the Trusted Node Dao to verify the signatures, introducing significant delays).

When BLS precompiles become available, upgrade `RocketMinipoolDelegate.stake()` to verify the signature.

Should an invalid deposit be made, ensure the Trusted Node watchtower software can correctly report the balance as lost. Ensure any trusted node members are carefully vetted and consider RPL bond requirements when choosing the unbonded validator limit.

Resolution

As this issue can not be feasibly actioned, the development team has closed this issue until BLS precompiles become available for EVM contracts.

RP-04	Trusted DAO Member Multiple Vote via Replace		
Asset	`rocketpool: RocketMinipoolStatus.sol`, `RocketNetworkPrices.sol` & `RocketNetworkBalances.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

Trusted DAO members perform an important service for the Rocket Pool network in acting as an oracle to report RPL price, eth2 balances and validator exit status. While new DAO members are prohibited from voting on previously existing proposals, there is no such restriction for submitting reports for prices, balances or withdrawables.

This can lead to a situation where a single entity repeatedly replaces their Trusted Node account and makes multiple malicious submissions (votes).

The impact is somewhat limited for price and balance reporting, as submissions must be for blocks more recent than the latest agreed-upon value. Given the proposal delays involved with replacing a DAO member, malicious submissions should usually be eclipsed by valid reports before enough malicious submissions can be amassed to reach the consensus threshold. However, because there is no restriction to prevent a submission for a future block, this is still a concern.

For withdrawable reporting, the potential risk is more serious. As shown in `test_malicious_withdrawal_reporting`,[2] it is technically possible for a single entity to make multiple successful transactions executing `RocketMinipoolStatus.submitMinipoolWithdrawable()` such that an excessive `_stakingEndBalance` is accepted and nETH minted such that the node operator can drain all ETH held in the nETH contract.

While it is readily possible for DAO members to identify this malicious behaviour and kick the offending members, that requires active monitoring and intervention. It is preferable to prevent this abuse in the first place.

While the testing team acknowledge that the proposal cooldown can provide some mitigation, that cooldown has no effect on a newly joined/replaced DAO member.

Recommendations

In `RocketNetworkPrices.submitPrices()` and `RocketNetworkBalances.submitBalances()` a require statement could be added that ensures `_block < block.number`, to protect against malicious submissions set far into the future allowing sufficient time to accumulate submissions before they are eclipsed by valid reports.

Ensure that `test_malicious_withdrawal_reporting` (or an equivalent test) passes (see the accompanying test framework delivered with this report).

Protect against duplicate withdrawable submissions by preventing new DAO members from reporting on previously existing submissions, or invalidate non-finalized submissions when members leave the DAO.

In practice, it is difficult to feasibly track DAO member "heritage", or to loop through and invalidate any non-finalized submissions when a DAO member is removed. This is because any unbounded looping may consume

[2] See Test Suite. The testing implementation is provided to the development team along with this report.

gas such that the transaction cannot be processed in a single block. Possible solutions could involve:

- Recording the block number for new withdrawable submissions, and comparing it to the block when the node became a member of the Trusted Node DAO.
- Enforcing a reporting window, such that this window is comparable to the time required to propose, replace, and activate new DAO members. Where the window limits the time for which a submission counts towards the consensus threshold, or prevents additional submissions after the window has expired.

Resolution

This was resolved for balance and RPL price reporting in commit 7fb9b52, that ensures network balances cannot be reported for a future block. A narrow time window is enforced such that, even with new and malicious members, fraudulent reports should not reach a significant threshold before they are invalidated.

The Trusted DAO membership replace functionality was removed in commits 17088c0 and 15a643d, so two proposals must be accepted in order to "replace" a member, increasing the barrier for amassing fraudulent reports.

Although it may be technically possible to accumulate fraudulent `RocketMinipoolStatus.submitMinipoolWithdrawable()` submissions, this is a similar threat to obtaining a controlling threshold of members, which is carefully managed. Incorrect `MinipoolWithdrawableSubmitted` events can be easily detected by other members.

RP-05	Less Secure Node Account Can Set Withdrawal Address		
Asset	`rocketpool: RocketNodeManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The node operator's account can always set the withdrawal address. Because the node account must make regular on-chain transactions (e.g. in order to claim RPL staking rewards), it is likely the withdrawal address (should it be different) is set to some account more highly trusted and secure than the node account (e.g. a hardware cold wallet).

It is sub-optimal to allow the less secure account to arbitrarily change the withdrawal address, particularly as the withdrawal address is used for all minipools associated with the node, not just new ones.

Recommendations

Consider requiring that the sender of a transaction executing `RocketNodeManager.setWithdrawalAddress()` must be the node's existing withdrawal address, to prevent redirection of funds should the node account become compromised.

To protect against accidentally setting the withdrawal address to the wrong value (and becoming unrecoverable), the following may be worth considering:

- Also allow the node account to set the withdrawal address, but only after a significant delay (during which the withdrawal account can cancel the change).
- Require that the new withdrawal account accept the change, similar to the "safe ownership transfer pattern".
- The "safe transfer" may be undesirable for some node operators who want to avoid making transactions from the withdrawal account whenever possible, so it could be useful to add a boolean flag to `setWithdrawalAddress()` that allows the address to be directly set without confirmation by the new withdrawal account.
- Alternatively, but similarly, the existing functionality could be kept by default, but a new "lock" parameter allows node operator to disable modification by the node address.

Resolution

This was resolved in commit c2f2308 requiring that any changes to a node's withdrawal address be sent from the current withdrawal address. It also allows node operators to perform a "safe ownership transfer", to protect against accidentally setting an invalid withdrawal addresses.

RP-06	Delayed Refunds for “FullDeposit” Minipools		
Asset	`rocketpool: RocketMinipoolQueue.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

Any available “half deposit” minipools will always be assigned funds before “FullDeposit”. While prioritizing `HalfDeposits` can maximize the amount of active Eth2 validators, in some circumstances, the refund for a `FullDeposit` minipool can be delayed for quite some time (theoretically indefinitely).

Recommendations

Two possible suggestions that could resolve this issue are:

- Occasionally prioritize `FullDeposit` over `HalfDeposit`. This is difficult to do randomly on-chain, so a “round-robin” method with a counter could be needed (e.g. every 4 deposits, a `FullDeposit` is prioritized).
- Have full and half deposits in the same queue.

Resolution

This has been acknowledged by the development team but is not an observed issue.

RP-07	Unmarshalling Pointers as `nil`		
Asset	`smartnode: shared/services/`		
Status	Open		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The default Golang JSON marshaller allows for nillable types such as `slices` and `pointers` to be unmarshalled as `nil`.

Within `RocketPool-CLI`, these `nil` objects are often dereferenced without first performing validity checks. Dereferencing a `nil` address will cause the program to panic.

Some examples of where nil-pointer panics may occur, after unmarshalling, are:

- In `smartnode/rocketpool-cli/node/status.go` the following lines:
  - line [33] `math.RoundDown(eth.WeiToEth(status.AccountBalances.ETH), 6)`
  - line [34] `math.RoundDown(eth.WeiToEth(status.AccountBalances.RPL), 6)`
  - line [35] `math.RoundDown(eth.WeiToEth(status.AccountBalances.NETH), 6))`
- In `smartnode/rocketpool-cli/node/swap-rpl.go` the following line:
  - line [33] `amountWei := status.AccountBalances.FixedSupplyRPL`
  - line [51] `entireAmount := status.AccountBalances.FixedSupplyRPL`
- In `smartnode/rocketpool-cli/auction/status.go` the following lines:
  - line [30] `math.RoundDown(eth.WeiToEth(status.TotalRPLBalance), 6)`
  - line [31] `math.RoundDown(eth.WeiToEth(status.AllottedRPLBalance), 6)`
  - line [32] `math.RoundDown(eth.WeiToEth(status.RemainingRPLBalance), 6))`

These panics occur when unmarshalling API response data. The impact is that CLI commands may not execute all of the desired transactions leaving the node or minipool in an undesirable state. However, this data comes from the a trusted source and is unlikely to provide malicious packets except by malfunction or misconfiguration (e.g. using different versions).

Recommendations

Consider recursively adding nil-pointer checks after unmarshalling in `smartnode/shared/services/` to ensure all struct fields (and if a field is a struct, array or map, then those fields recursively) do not contain `nil` unless it is strictly necessary.

RP-08	SSH Passphrase as Command Line Argument		
Asset	`smartnode: RocketPool-CLI`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The passphrase for SSH connections is passed as a command line argument. It is recommended against allowing passphrases to be passed as command line arguments as they will appear in the process table and shell history.

As the current threat model assumes that the RocketPool-CLI is run from a fairly trusted environment, this issue is classified as low impact.

Recommendations

We recommend using a flag which will cause the passphrase to be read as standard input to be the ***default*** method for handling passwords. Passwords should be read using the Golang function `terminal.ReadPassword()`.

As a password prompt is troublesome for non-interactive use-cases, consider replacing the current CLI flag with a parameter that accepts a path to a file containing the secret. This can be more secure, as file permissions can prevent access by other users, and the files can be saved to memory-backed partitions (so are never written to disk). This is compatible with `docker compose` secrets [3].

Also consider renaming the CLI flag to something containing the word “unsafe”, to discourage uninformed use.

Recommend, in documentation, that the node operators run their software on dedicated VMs or machines without any untrusted user accounts.

Resolution

This was resolved in commit 4e227ff by reading from a passphrase file instead of a command line argument.

RP-09	Insufficient Password Strength & Complexity Requirements		
Asset	`smartnode: shared/services/passwords/manager.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Passwords are used to encrypt keystores to prevent malicious users from reading the private key data. This includes both on-chain and off-chain keys.

There is only one password strength requirement when creating passwords, that is a minimum length of eight (8) characters.

Some users may create weak keys which can be brute forced or easily guessed, thereby exposing private key information.

This vulnerability is considered a *low* severity as the password file is stored inside the docker container along with the encrypted keystore, and is assumed to be run in a fairly trusted environment. It is likely that anyone with access to the keystore (in the default installation) will also have access to the password file.

Recommendations

Consider implementing and enforcing a strong password policy:[3]

- Passwords should be 12 characters or longer.
- Prospective passwords should be checked against a blacklist of commonly used passwords.

Resolution

This was resolved in commit 380f8ec by updating the minimum password length from 8 to 12.

[3] See also NIST guidelines: 5.1.1.2 Memorized Secret Verifiers — `https://pages.nist.gov/800-63-3/sp800-63b.html#sec5`

RP-10	Panic in the Unmarshalling of `ValidatorPublicKey` & `ValidatorSignature`		
Asset	`rocketpool-go: types/beacon.go`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The structs `ValidatorPublicKey` and `ValidatorSignature` in `rocketpool-go` are fixed size byte arrays of length 48 and 96 respectively. These structs both implement the function `UnmarshalJSON()` which takes a string of hex characters and converts it to bytes using the Golang library `encoding/hex` .

The function `hex.Decode([]byte, string)` will panic if the length of the bytes that are decoded (`length(string) / 2`) are larger than the byte array provided. Hence, in the example below if the function `HexToValidatorPublicKey()` is passed a hex string of even length greater than `ValidatorPubkeyLength * 2` , it will panic.

```go
func HexToValidatorPubkey(value string) (ValidatorPubkey, error) {
    pubkey := make([]byte, ValidatorPubkeyLength)
    if _, err := hex.Decode(pubkey, []byte(value)); err != nil {
        return ValidatorPubkey{}, err
    }
    return BytesToValidatorPubkey(pubkey), nil
}
```

A similar issue occurs in the function `HexToValidatorSignature()` as seen below.

```go
func HexToValidatorSignature(value string) (ValidatorSignature, error) {
    signature := make([]byte, ValidatorSignatureLength)
    if _, err := hex.Decode(signature, []byte(value)); err != nil {
        return ValidatorSignature{}, err
    }
    return BytesToValidatorSignature(signature), nil
}
```

Recommendations

We recommend ensuring the length of the string to be converted to a signature or public key is exactly twice the required length (`2 * ValidatorPubkeyLength` and `2 * ValidatorSignatureLength` respectively) else returning an error.

Resolution

The function, `EncodedLen()` , is used to return the length of the encoded src bytes, which is effective twice the input bytes. This was resolved in commit 5091812 by enforcing a length check between the `value` string and `EncodedLen(dst)` .

RP-11	Incorrect Access Control List for `_upgradeContract()` Function		
Asset	`rocketpool: RocketDAONodeTrustedUpgrade.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Currently, the Rocket Pool Trusted DAO restricts upgrades to contracts designed to be immutable. These include: `rocketVault`, `rocketPoolToken`, `rocketTokenRETH`, `rocketTokenNETH` and `casperDeposit`. However, the `rocketPoolToken` contract does not exist within Rocket Pool's storage. The testing team believes the naming of this contract has changed previously and instead should be `rocketTokenRPL`.

Recommendations

Consider changing `rocketPoolToken` in `RocketDAONodeTrustedUpgrade.sol:45` to `rocketTokenRPL`.

Define all contract names as constants in a central abstract contract that all Rocket Pool contracts import. This avoids any risk of inconsistent strings and typos.

Consider sourcing these names in the contract deployment script or, at minimum, checking that the property names of the `contracts` object defined in `2_deploy_contracts.js` have matching constants.

Resolution

The recommendation has been implemented by the development team in commit d0fc258.

RP-12	Network Contracts Have Unrestricted Access to Storage		
Asset	`rocketpool: RocketStorage.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `RocketStorage` contract is used for centralized data storage, allowing other contracts to be replaced without migrating data. Any registered network address has full access to modify any storage location, including those used by other contracts. This is an arguably excessive attack surface, where complete compromise of a single contract results in that of the entire network.

This also weakens the effectiveness of other "defense in depth" measures. For example, `RocketVault` only allows network contracts to withdraw their own funds (as determined by `"contract.name"`).

However, a malicious network contract can simply change its `"contract.name"` to `"rocketDepositPool"` and withdraw all associated funds.

Similarly, a network contract can set `"contract.storage.initialised"` to false, thus allowing anyone to make arbitrary storage modifications, when the apparent intention is for this to only be changed during initial deployment.

The testing team *note* that it is more unlikely that a vulnerable contract can be exploited to make arbitrary storage changes. This can be considered analogous to a conventional "remote code execution" vulnerability, as opposed to injection of specific values.

See also the related issue RP-31.

Recommendations

Consider using a standalone state variable to represent `"contract.storage.initialised"` , to prevent registered network contracts from later unsetting the value via `setBool()` or `deleteBool()` . This also has some added performance benefits (see RP-18).

Consider implementing more granular access control for `RocketStorage` . This could involve the `RocketStorage` enforcing restricted "key-prefix" namespaces, where only certain contracts have write access to relevant namespaces. For example, it would be of benefit to allow only `RocketDAONodeTrustedUpgrade` the ability to modify storage values with keys beginning with `"contract.exists"` , `"contract.name"` , `"contract.address"` , or `"contract.abi"` . A more flexible implementation could be for `RocketStorage` to only allow modifications via a single (upgradable) ACL contract.

The testing team acknowledges that this granular access control would likely require significant tradeoffs with regards to upgrade flexibility and gas costs. Enforcing key restrictions via namespace would likely require passing string keys to `RocketStorage` for validation, thus making it more complicated but not impossible to update or requiring the gas costs of an additional layer of abstraction via an upgradable ACL contract.

As access control improvements could well involve significant re-architecting and introduce other drawbacks, it can be reasonable to mitigate these risks through alternative means. These could include:

- Ensuring all development team members and contributors are aware that any contract changes have the potential to affect the operation of unrelated and highly critical contracts (e.g. via CONTRIBUTING documentation and checklists in Pull Request templates).
- Ensuring DAO members with the ability to vote on protocol upgrades are similarly aware that the registration of a single corrupt or malicious contract can result in catastrophic impacts including loss of funds.

  Those currently responsible are members of the Trusted Node DAO, later to be shifted to Protocol DAO members.

  This awareness could be fostered via DAO member documentation and reminder checklists/warnings in the voting UI.
- Enforcing a careful and staged upgrade process, where delays are enforced between an upgrade proposal's creation, vote, and when it takes effect.
- Centrally document the namespaces used for storage keys, to avoid situations where unrelated contracts accidentally use the same key for different purposes.

Note: because these involve changes to the underlying `RocketStorage` contract, they cannot be applied as an upgrade to an existing Rocket Pool network.

Resolution

This has been acknowledged by the development team and no technical mitigations are deemed actionable due to excessive gas costs.

RP-13	Divide before Multiply		
Asset	`rocketpool: RocketAuctionManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Solidity is unable to handle floating point numbers and will therefore truncate the integer during division. This may result in an unnecessary loss of precision when attempting to calculate lot prices at a specific block in `RocketAuctionManager.getLotPriceAtBlock()`. The current implementation involves `mul(tn).div(td).mul(tn).div(td)` when, ideally, all multiplication should be done prior to any division (provided overflow is not a risk).

However, due to high precision of `ether` values, this issue is of low severity.

Recommendations

The testing team recommends performing all multiplication operations before any division whenever possible.

Resolution

The recommendation has been implemented by the development team in commit 05a24ac.

RP-14	Refund of Successful Challenge		
Asset	`rocketpool: RocketDAONodeTrustedActions.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Rocket Pool's Trusted Node DAO contains a set of trusted DAO members who are invited by current members of the DAO and are required to stake a significant RPL bond. If enough of these members go offline and stop responding to proposals, it's possible that no more proposals could be passed. As a result, Rocket Pool includes a mechanism that allows regular nodes to challenge trusted DAO members. If the member does not respond within a given window, they are removed from the trusted DAO and their RPL bond is not returned.

To submit a challenge, a regular node must pay some ETH (to prevent spam). However, this ETH is not returned to the challenger if the challenge is found to be successful.

Recommendations

The testing team recommends refunding the ETH value used to challenge a trusted DAO member in `RocketDAONodeTrustedActions.sol.actionChallengeDecide()`.

Although a successful challenger would likely be rewarded out-of-band for their meritorious service, it may be worthwhile to explicitly reward the challenger with a portion of the kicked member's RPL bond.

Additionally, the ETH sent to the contract could be better utilised by sending it to the `RocketVault`, should a challenge be unsuccessful.

Resolution

The development team decided it is safer to burn ETH that is used to challenge other trusted DAO nodes in order to remove all opportunities to game the challenge mechanism.

RP-15	Inaccurate calculation of `getTotalEffectiveRPLStake`		
Asset	`rocketpool: RocketNodeStaking.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Calculations for `RocketNodeStaking.getTotalEffectiveRPLStake()` will produce unwanted results when some nodes have more than the *150%* capacity (`node.per.minipool.stake.maximum`) but the network as a whole has not yet achieved the maximum percentage.

In the following, the calculation acts as if the RPL distribution is evenly balanced throughout all nodes.

```
uint256 maxRplStake = rocketDAOProtocolSettingsMinipool.getHalfDepositUserAmount()
.mul(rocketDAOProtocolSettingsNode.getMaximumPerMinipoolStake())
.mul(rocketMinipoolManager.getMinipoolCount())
.div(rocketNetworkPrices.getRPLPrice())
```

While it is true that `maxRplStake` correctly reflects the amount of RPL corresponding to when all node operators stake the maximum RPL, this can be larger than the *actual* effective RPL total when some nodes stake over 150% but others don't.

As an example, consider a Rocket Pool network consisting of 2 node operators (each with a single minipool). Node A stakes the minimum 10% (1.6 ether worth of RPL), and Node B stakes 200% (32 ether worth of RPL). As such, the results for `getNodeEffectiveRPLStake()` are equivalent to 1.6 and 24 ether respectively (with Node B limited to 150%). This gives a sum equivalent of 25.6 ether. However, the current `getTotalEffectiveRPLStake()` implementation only limits the stake to a maximum of $150\% \times 2 \times 16 = 48$ ether worth of RPL. As `getTotalRPLStake()` returns 33.6 ether worth of RPL, `getTotalEffectiveRPLStake()` returns the full 33.6, counting *all* of Node B's stake towards the total.

From another perspective, Node B's funds are truncated in the numerator but not the denominator, resulting in a smaller share of total rewards.

As can be seen, `getTotalEffectiveRPLStake()` truncates to an upper bound, but not an exact value. The impact is primarily felt in `RocketClaimNode.getClaimRewardsPerc()` , which is used to distribute RPL rewards proportional to each node's share of the total effective stake (i.e. $\propto$ `getNodeEffectiveRPLStake(node).div(getTotalEffectiveRPLStake())`). When `getTotalEffectiveRPLStake()` returns a larger value, each node receives a smaller share of claim rewards.

For the example above, it is as if there is an extra 3rd node operator staking 8 ether worth of RPL. These remaining claim rewards are left in the rewards pool to be distributed in subsequent claim intervals. Although the original recipients will receive some of their funds in later intervals, some will also be distributed to Trusted Node DAO members, the Protocol DAO treasury, and new node stakers.

While losses from this inaccuracy can be thought to be distributed proportionally across all node operators, that is not actually true. Given gas fee overheads comprise a comparatively larger portion of rewards for smaller RPL stakers, this can understood to disproportionately affect them. That is, small nodes have lower profit margins, so any losses affect them more.

This may effect the validity of economic incentive analysis, used to ensure that it is profitable for small nodes to pay the gas fees to claim every interval. There may even be incentive for RPL whales to stake well over 150% in order to make it unprofitable for small stakers to claim, thus leaving their rewards to be distributed in subsequent rounds. (As allocated rewards must be claimed within the ≈ 14 day interval or they are lost.)

Recommendations

Consider adjusting `getTotalEffectiveRPLStake()` such that it returns the equivalent of

```
total = 0
for node in all_registered_node_operators:
    total += getNodeEffectiveRPLStake(node)
return total
```

It may be infeasible/unreasonable to implement this in a gas-efficient manner (if gas costs are proportional to the number of nodes, there could be risk of exceeding the block gas limit).

In that case, it may be necessary to leave the current implementation unchanged. Then, the testing team recommends that this inaccuracy is clearly documented, and node operators are informed that staking rewards should be expected to reduce slightly when some nodes stake more than the maximum (this is still distributed in subsequent claim intervals).

If possible, try to stop node operators from being able to affect the scale of this inaccuracy.

Consider this in any analysis of economic incentives to ensure the relevant properties still hold, and adjust fee distribution values as needed. Because the scale of this inaccuracy can be controlled by node operators, it may be difficult to predictably model.

Resolution

This issue has been acknowledged by the development team but no feasible technical fix could be identified, hence this is marked *closed*.

RP-16	Reliance on ETH1 Provider		
Asset	`rocketpool-go`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Rocket Pool contacts are stored centrally on-chain in the `RocketStorage` contract. When the `rocketpool-go` library is interacting with any contracts other that the `RocketStorage` contract these are retrieved on-chain.

ETH1 providers may be run locally as Geth nodes or users may select third parties to provide ETH1 services. The following is a potential vulnerability related to using untrusted ETH1 providers.

If the ETH1 provider were to deliberately return a malicious address from an `eth_call`, they may have users execute transactions to the wrong contract.

For example if a malicious ETH1 provider were to modify the address returned for the call `RocketStorage.getContract('rocketNodeDeposit')`, the node may execute a deposit that transfers 16 or 32 ETH to a malicious contract owned by the ETH1 provider.

Recommendations

Ensure that node operators are aware of the security reliance when using third party ETH1 providers.

Resolution

This has been acknowledged by the development team and has determined not technically actionable.

RP-17	Ineffective RPL Staking Collateral
Asset	`rocketpool: RocketMinipoolStatus.sol`
Status	Open
Rating	Informational

Description

Nodes operators stake RPL intended for use as collateral that can be sold by the protocol to protect rETH holders from any losses in their balance. The current implementation makes use of the RPL collateral *only* after all ETH collateral is exhausted. Given current penalty calculations, it is highly unlikely that a validator could exit with less than 16 ETH (even when slashed). As such, the RPL collateral is unlikely to ever be used, even for very mediocre node operators who do not increase staking rewards.

While there is some technical risk that an honest validator could be slashed — that a bug in the client can cause a slashable offence — this is quite minor. Similarly, there is some security risk that the validator's key could be compromised, allowing a malicious party to perform a slashable offence, though this can be reasonably mitigated by a conscientious node operator. The main cause of a slashable offence is due to ignorant or malicious actions on the part of the validator.

There is also some external risk with regards to inactivity penalties that cannot be fully removed (e.g. network disconnection due to natural disaster or other infrastructure failure), but the penalties are such that it is highly unlikely that this would ever result in a balance decrease over a reasonable staking period.

As such, the main risk of any staking losses is towards the Rocket Pool network from a malicious or negligent node operator. There is little risk to an honest node operator such that it should be reflected in RPL staking rewards. Instead, the rewards should reimburse the operator for opportunity costs associated with locking up value in RPL.

Given that slashing penalties are currently on the order of only ≈ 2 ETH in normal network conditions, a bonded validator should always have enough ETH balance remaining to cover the User's original 16 ETH deposit, even if the validator is extremely negligent.

Recommendations

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

Confirm current Eth2 staking penalty calculations and include them in any decisions regarding RPL staking rewards and penalties.

It may be worthwhile penalising exited validators with a balance less than 32 ETH (less than they started with), even though rETH has no danger of losing value. By slashing some RPL from these validators, the protocol is recovering some lost profits that it should expect from a conscientious validator. It is not unreasonable to expect that an exited validator should return some profit.

One implementation could involve slashing RPL proportional to the deficit in the final balance (amount below the 32 ETH starting balance).

RP-18	Likely Gas Savings When Setting RocketStorage Values
Asset	`rocketpool: RocketStorage.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `RocketStorage.onlyLatestRocketNetworkContract()` modifier can be implemented in a more gas-efficient manner, resulting in widespread gas savings.

The `RocketStorage` contract is used for centralized data storage, allowing other contracts to be replaced without migrating data. To prevent unauthorized storage modification, every `set*()` and `delete*()` function has the `onlyLatestRocketNetworkContract` modifier. To allow direct modification during deployment, `onlyLatestRocketNetworkContract` checks for a boolean flag that is set upon deployment completion (see below).

```
if (boolStorage[keccak256(abi.encodePacked("contract.storage.initialised"))] == true) {
```

Two `keccak256` operations are performed in order to access this boolean value:

1. To determine the mapping key (the result of `keccak256(abi.encodePacked("contract.storage.initialised"))`).
2. To determine the location of the value in storage (`keccak256(uint256(4) . key)`).[4]

Although the gas used for these `keccak256` operations is relatively small (on the order of of 36 gas per operation[4]), this is highly trafficked code and thus has a comparatively larger effect on the deployment and normal operation of the Rocket Pool network contracts.

Because the key is fixed, it is possible to write an equivalent check that requires no `keccak256` operations like so:

```
// NOTE:
// boolStorage is at slot 4
// ‘keccak256(uint256(4) . keccak256("contract.storage.initialised"))‘ evaluates to the
// hexnumber below
assembly{
  let isInit := sload(0xd58b0b884eb0b2eb26bfe63b69fb2d6af87f743960a008c65a1ab0385746612a)
  if eq(isInit, true) {
  ...
}
```

While this is possible to do using inline assembly, it is clearer and less error prone to simply store the initialised flag as a standalone state variable (and thus stored at a small-integer storage location).

[4] See https://docs.soliditylang.org/en/v0.7.6/internals/layout_in_storage.html#mappings-and-dynamic-arrays

As of testing, the `solc` optimizer does not compute `keccak256` at compile-time, so these optimizations can only be implemented by a programmer.[5]

Recommendations

Consider using a standalone state variable to represent `"contract.storage.initialised"`, as described above.

This also has some security benefit in preventing a registered network contract from later setting `"contract.storage.initialised"` to `false` (see RP-12).

NOTE: because this is a change to the underlying `RocketStorage` contract, it cannot be applied as an upgrade to an existing Rocket Pool system.

Resolution

The development team has resolved this issue by updating the line of code below from

```
if (boolStorage[keccak256(abi.encodePacked("contract.storage.initialised"))] == true) {
```

to

```
if (storageInit == true) {
```

By using state variables to store the deployed status of the `RocketStorage` contract, the end-user is able to save a small amount of gas each time they interact with Rocket Pool's smart contracts.

These changes are reflected in commit 1879fac.

[5] For instance, moving the `keccak256(abi.encodePacked("contract.storage.initialised"))` key to a private constant has no effect. (Though using a precomputed key of `0x1a655af42e38e46646ca444968abc315a08696908ac9b25256e67e1a25f98eb4` would be an improvement.)

RP-19	Gas Savings via Bulk and Update Storage Functionality
Asset	`rocketpool: RocketStorage.sol`
Status	Open
Rating	Informational

Description

The `RocketStorage` contract is used for centralized data storage, allowing other contracts to be replaced without migrating data, and as a registry of relevant contract addresses. While this pattern offers great flexibility and avoids complicated proxy patterns, any storage access incurs the additional gas overheads involved with calls to external contract functions. With a few additional features, it should be possible to greatly reduce the number of external function calls performed.

Less gas overheads for node operators and DAO members can make it reasonable to reduce or divert RPL inflation rewards without affecting the validity of economic incentives.

The identified features are described in more detail below, but both focus on minimizing cases where multiple, consecutive calls to `RocketStorage` are required.

Update Operations

A common use of `RocketStorage` is to adjust counter or balance values. Currently, this is done via a "get" operation followed by a "set", like in `RocketNodeStaking` (below):

```
function getTotalRPLStake() override public view returns (uint256) {
  return getUintS("rpl.staked.total.amount");
}
function setTotalRPLStake(uint256 _amount) private {
  setUintS("rpl.staked.total.amount", _amount);
}
function increaseTotalRPLStake(uint256 _amount) private {
    setTotalRPLStake(getTotalRPLStake().add(_amount));
}
function decreaseTotalRPLStake(uint256 _amount) private {
    setTotalRPLStake(getTotalRPLStake().sub(_amount));
}
```

Here, the value returned by `getUintS` is *only* used for the subsequent set operation, and is otherwise unneeded by the calling contract.

As such, implementing an arithmetic update operation like `RocketStorage.addUintS()` will allow an equivalent `increaseTotalRPLStake` to make only a single external call:

```
function increaseTotalRPLStake(uint256 _amount) private {
    addUintS("rpl.staked.total.amount", _amount);
}
```

Bulk/Batch Operations

Almost all Rocket Pool contract functions begin with a series of `RocketStorage.getContractAddress()` calls, to identify the current addresses of the relevant Rocket Pool contracts. These can be condensed into a external function call, which can noticeably reduce gas costs.

Similarly, many functions contain a series of consecutive storage modifications, each making an external call to `RocketStorage`. While it is more complicated to bundle these modifications for heterogeneous types, it is comparatively simple to group "setters" for the same type. This would have a noticable gas saving in high traffic functions like `RocketRewardsPool.claim()`, where the 9 `setUint` calls involving the following keys could feasibly be bundled into a single external `bulkSetUint()` call:

- `"rewards.pool.claim.interval.total"`
- `"rewards.pool.claim.interval.block.start"`
- `"rewards.pool.claim.interval.contract.perc.current"`
- `"rewards.pool.claim.interval.contract.total"`
- `"rewards.pool.claim.interval.contract.perc.current"`
- `"rewards.pool.claim.interval.contract.allowance"`
- `"rewards.pool.claim.interval.claimers.total.current"`
- `"rewards.pool.claim.interval.contract.total"`
- `"rewards.pool.claim.interval.block.last"`

Recommendations

Consider implementing additional `RocketStorage` functionality that allows batch and/or update operations, to reduce the number of external calls made during deployment and, therefore, gas costs. Refactor existing code to take advantage of these operations.

NOTE: because this involves changes to the underlying RocketStorage contract, it cannot be applied as an upgrade to an existing Rocket Pool system.

Update Operation Recommendations

As not all relevant code is as easily refactored as `increaseTotalRPLStake()`, a more flexible interface would be to implement update functions in the form of

```
function addUint(bytes32 _key, uint _value) external returns (uint _result);
```

Where the value returned is either result of the operation or the original value at `_key` (similar to an atomic "fetch-and-add" instruction[6]). Returning a value allows use in more situations where the calling code makes some associated `require` check.

[6] See https://en.wikipedia.org/wiki/Fetch-and-add

Given that `RocketStorage` is a trusted contract, this refactoring is not expected to introduce any re-entrancy vulnerabilities.

Also implement similar functions for the `uint` and `int` types for the `add`, `sub` operations (and potentially `mul` and `div`)

Bulk/Batch Operation Recommendations

Strongly consider implementing bulk operation functions for `RocketStorage`.

At minimum, a bulk address "getter" will have a noticeable reduction in the number of external calls required. Such an operation could look like:

```
function getAddresses(bytes32[] calldata _keys) external returns (address[] memory);
```

Consider implementing bulk "getter" and "setter" functions for the various storage types. Although these may not necessarily be used as much in the current code, they provide "future proofing". Such functions could look like:

```
function bulkGetUint(bytes32[] calldata _keys) external returns (uint[] memory);
function bulkSetUint(bytes32[] calldata _keys, uint[] calldata _values) external;
function bulkSetBool(bytes32[] calldata _keys, bytes calldata _bits) external;
```

Also consider bulk operations for heterogeneous types. Evaluate whether any gas savings involved are sufficient to justify the extra programming complexity involved with "manual" abi-encoding. Such an interface could look like the following:

```
enum StorageType { Address, Uint, String, Bytes, Bool, Int, Bytes32 }
function bulkGetStorage(bytes32[] calldata _keys, StorageType[] calldata _types) external
        returns (bytes[] memory _encodedValues);
function bulkSetStorage(bytes32[] calldata _keys, StorageType[] calldata _types, bytes[]
        calldata _encodedValues) external;
```

RP-20	RocketTokenRPL.swapTokens gas savings
Asset	rocketpool: RocketTokenRPL.sol
Status	**Resolved:** See Resolution
Rating	Informational

Description

Redundant checks in `swapToken` can lead to increase gas cost of the function call. Although providing more user-friendly error messages, a number of these checks are performed as part of the base `rplFixedSupplyContract`, proving redundant as they are integrated as part of the ERC20 token checks.

- At line [193] `require(rplFixedSupplyContract.balanceOf(address(msg.sender)) > 0)` is also checked as part of `ERC20._transfer`.
- At line [195] `require(rplFixedSupplyContract.balanceOf(address(msg.sender)) > _amount)` makes the above check redundant, and is also checked as part of `ERC20._transfer`.
- At line [199] The requirement for `allowance` is also checked as part of `ERC20._transferFrom`.

Recommendations

The testing team recommends considering the tradeoffs between user-friendly revert messages, and gas savings.

As these requires are performed as part of `ERC20` base functionality, they can be omitted from the `RocketTokenRPL` contract for gas savings during runtime and deployment.

Resolution

The development team has removed the redundant checks found in the `swapTokens()` function. This can be found in commit d8667dc.

RP-21	Unhandled Errors
Asset	`rocketpool-go` & `smartnode`
Status	**Closed:** See Resolution
Rating	Informational

Description

A range of errors in go are not handled correctly. These errors have been deemed as informational, as no exploit over these errors could be identified.

Related code in `rocketpool-go` :

```
// rocketpool-go/rocketpool/abi.go:52
  51:     }
> 52:     zlibWriter.Flush()
  53:
```

Related code `smartnode`

```
// smartnode/shared/services/wallet/validator.go:207
  206:     initBLS.Do(func() {
> 207:         eth2types.InitBLS()
  208:     })
```

```
// smartnode/shared/services/rocketpool/client.go:185
  184:             if verbose {
> 185:                 c.Println(scanner.Text())
  186:             }
```

```
// smartnode/shared/services/rocketpool/client.go:108
  107:     if c.client != nil {
> 108:         c.client.Close()
  109:     }
```

```
// smartnode/shared/services/config/config.go:120
  119:     for i := len(configs) - 1; i >= 0; i-- {
> 120:         mergo.Merge(&merged, configs[i])
  121:     }
```

```
// smartnode/shared/services/beacon/prysm/client.go:53
  52: func (c *Client) Close() {
> 53:     c.conn.Close()
  54: }
```

```
// smartnode/rocketpool-cli/wallet/init.go:65
  64:     // Clear terminal output
> 65:     term.Clear()
  66:
```

Recommendations

Consider propagating or handling the errors for each of these cases.

Consider integrating errcheck and other security linters into the CI workflow.

Resolution

This issue is outdated and does not match the latest commit targets. As a result, this issue has been closed in favour of `RP-36`.

RP-22	Unused and Lack of Constant Variables
Asset	`rocketpool: contracts/contract/*`
Status	Open
Rating	Informational

Description

There are a number of variables used throughout the Rocketpool smart contract system that can be specifically declared as constant variables using the `constant` keyword. When declared as state variables, these require additional gas costs to set and access from storage.

Additionally, there are some variables that are unused and therefore can be omitted.

- `RocketBase.version` should be defined as a constant, or `immutable` if it's desirable that the value can be set during construction.
- Omit `calcBase` in `RocketDAONodeTrusted.sol` as it is unused.
- Define `calcBase` as a constant wherever it is used.

  In particular, replace the following `calcBase` definitions:

  - `RocketClaimNode.sol` line [50]
  - `RocketClaimTrustedNode.sol` line [17]
  - `RocketRewardsPool.sol` line [19]
  - `RocketNetworkFees.sol` line [47]
  - `RocketNodeStaking.sol` line [187]
  - `RocketNetworkBalances.sol` line [63,93]
  - `RocketDAOProposal.sol` line [23]
  - `RocketTokenRPL.sol` `10^18` in `inflationCalculate()` lines [139,144]
  - `RocketDAONodeTrustedProposals.sol` line [23]
  - `RocketTokenRETH.sol` line [73]
  - `RocketDAOProtocolProposals.sol` line [22]
  - `RocketDAOProtocolActions.sol` line [20]
  - `RocketNetworkPrices.sol` line [68]
  - `RocketMinipoolStatus.sol` lines [61,118]
  - `RocketAuctionManager.sol` lines [137,152,174,217,248]
- `daoNameSpace` and `daoMemberMinCount` in `RocketDAONodeTrustedProposals`
- `totalInitialSupply` in `RocketTokenRPL`
- `RocketMinipoolDelegate.rocketStorage`, `RocketMinipool.rocketStorage`, and `RocketBase.rocketStorage` can be set as `immutable`.

Recommendations

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

Consider updating state variables to be declared `constant` whenever possible and omit any unused variables from contracts that do not require them. Both these changes will save some gas and improve maintainability.

Replace storage key string literals with centrally defined constants wherever possible.

RP-23	Consolidation of `RocketDAONodeTrustedActions`
Asset	`rocketpool: RocketDAONodeTrustedActions.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Below is a list of small changes that can be made to the `RocketDAONodeTrustedActions.sol` contract to add user clarity and maintain code consistency:

- The current standard is to call `getContractAddress` and cast it into its corresponding interface in the same line of code. i.e.

  ```
  RocketDAONodeTrustedInterface rocketDAONode = RocketDAONodeTrustedInterface(
          getContractAddress("rocketDAONodeTrusted"));
  ```

  However, `_memberJoin()` at lines [86,87] does not follow this same pattern and instead accesses the address from `RocketStorage`, then later casting this address into its corresponding interface on another line. This can be consolidated into a single line as previously mentioned.

- When a registered node attempts to join as a member with a sufficient RPL bond, it is unclear whether the user was invited in the first place. Consider checking if `memberInvitedBlock` is zero before checking if the invite was expired.

Recommendations

The testing team recommends implementing the above changes to `RocketDAONodeTrustedActions.sol`.

Resolution

The development team has decided not to implement the changes pertaining to the casting of addresses. An additional revert check has been added to ensure a node was invited to join the trusted DAO. This is implemented in commit f8bb4aa.

RP-24	Lack of Input Validation
Asset	`rocketpool: contracts/contract/dao/*`
Status	Open
Rating	Informational

Description

This section details instances where input validation would prove useful in preventing function misuse:

- `RocketDAONodeTrustedUpgrade:_upgradeContract()` does not check if the `_contractABI` value is an empty string before setting.
- `RocketDAONodeTrustedUpgrade:_upgradeABI()` does not check if the `existingAbi` is the same as parsed `_contractAbi`. Therefore, an event will be emitted for an ABI upgrade when no such upgrade occurred.

Recommendations

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

RP-25	Suboptimal Definition of MiniPool Storage Layout
Asset	`rocketpool: RocketMinipool.sol & RocketMinipoolDelegate.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `RocketMinipool` contract refers to `RocketMinipoolDelegate` via `delegatecall` for the majority of its functionality. Known as a "Delegate Proxy Pattern", this is useful to reduce minipool deployment costs but it is important that both contracts maintain a storage layout consistent with each other (see [5, section 4]). While the current implementations of RocketMinipool and RocketMinipoolDelegate define a consistent storage layout, this is done in a suboptimal manner, where mistakes can easily be introduced.

Both contracts define a storage layout (at lines [14-41] and [28-55] respectively) but, by defining the layout twice, the risk of "copy-paste errors" is introduced, and both definitions must be consistent to avoid unexpected bugs (particularly if the RocketMinipoolDelegate is upgraded in production). It is preferable to define the storage layout centrally, where both inherit from this definition.

Recommendations

Consider implementing a " `MinipoolStorageLayout` " contract (or equivalent) that defines all the state variables used by the `RocketMinipool` and `RocketMinipoolDelegate` contracts, then have both contracts inherit this definition.

After the Rocket Pool network is in production, this storage layout contract should be kept immutable. A good practice to enforce this is to assert, in CI, that the `MinipoolStorageLayout.sol` matches a known hash, so any changes are visible as failed tests.

Any subsequent upgrades that change the storage layout should be implemented in new contracts that inherit from the original (e.g. `contract MinipoolStorageLayoutv2 is MinipoolStorageLayout`). This ensures that new state variables are located at a later storage slots and the original storage layout is unchanged, so any existing minipool instances still function as expected.

Resolution

The development team has implemented the above changes in commit 1bfa2a7.

RP-26	DAO Settings Checks
Asset	`rocketpool: contracts/contracts/dao/protocol/settings/*`
Status	Open
Rating	Informational

Description

The testing team feels that a number of DAO related actions require careful consideration when transitioning from the guardians to a DAO.

The following items require additional checks to ensure safety for future DAO interaction:

- `RocketDAOProtocolSettingsInflation.sol`
  - `rpl.inflation.interval.rate` - If set less than `RocketTokenRPL.totalSupply()` then will cause a reverts on `RocketTokenRPL.inflationMintCalculate()`
  - `rpl.inflation.interval.blocks` and `rpl.inflation.interval.rate` are commented to be dependent on each other, but can be set separately.

Recommendations

The testing team recommends precautions when updating DAO settings as guardians, but also encourages additional checks and requirements when transitioning to a full DAO proposal/voting process.

RP-27	Rounding of Auction Bids
Asset	`rocketpool: RocketAuctionManager.sol`
Status	**Open**
Rating	Informational

Description

Rocket Pool's auction system is designed to cover the loss of ETH caused by slashed node operators. Users participate in a Dutch style auction to claim Rocket Pool Tokens (`RPL`) at a discount. These auctions are separated into lots which are then bid on. When a bid is placed on a lot, a portion of the lot is set aside based on the current lot price. The auction system performs some small rounding for each bid claimed. This can be seen in `RocketAuctionManager.sol` line [249]:

```
uint256 rplAmount = calcBase.mul(bidAmount).div(currentPrice);
```

Recommendations

The testing team recommends transferring the leftover wei to the last claimer of the lot, to simply avoid residual RPL remaining in the Auction contract.

RP-28	Improper Emitting of Events
Asset	`rocketpool: RocketDAOProposal.sol & RocketDAONodeTrustedActions.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `RocketDAOProposal.add()` and `RocketDAOProposal.execute()` functions emit the events `ProposalAdded` and `ProposalExecuted` respectively. Both these functions utilise `msg.sender` as the `proposer` or `executer` argument. If these functions are called via the `RocketDAONodeTrustedProposals` contract (which is the correct method of interacting with proposals), the `proposer` and `executer` arguments will be the address of the `RocketDAONodeTrustedProposals` contract instead of the intended `proposer` or `executer`.

Additionally, the event `ActionReplace` in `RocketDAONodeTrustedActions.actionReplace()` sets both the current member argument and the new member argument to the same address.

Recommendations

The testing team recommends changing the values of the events emitted in `RocketDAOProposal` to the trusted DAO member that called these functions in `RocketDAONodeTrustedProposals`. We also recommend changing the second argument in the emitted event in `RocketDAONodeTrustedActions.actionReplace()` to the member's address that is replacing the current member.

Implement relevant tests.

Resolution

The development team has implemented the recommended changes.

`RocketDAONodeTrustedActions.actionReplace()` has been removed altogether from the `RocketDAONodeTrustedActions` contract. See commit b8356dc for changes.

RP-29	Potential Settings “Getter” Gas Optimizations
Asset	`rocketpool: contracts/contract/dao/*/settings`
Status	Open
Rating	Informational

Description

Although the Rocket Pool settings design allows for flexible upgradability and management by the DAOs, accessing most settings values requires two function calls to external contracts.

Most `getXXX` settings “getter” functions involve access to value held in `RocketStorage`, so any contract accessing that setting first needs to make an external call to the relevant settings contract, which then calls `RocketStorage`. This can result in significant gas overheads, particularly for small, regularly executed functions that need to first get the relevant settings contract address from `RocketStorage`.

Should it be acceptable for the relevant storage keys to remain relatively fixed, the number of external calls required to access a setting can be reduced to a single external call to `RocketStorage`. This can be achieved by including the “getters” as local functions in the relevant contracts that use them.

The testing team acknowledge that this may hinder upgradability e.g. should it be desirable to modify the relevant setting storage keys. However, the runtime gas savings should be significant for frequently accessed settings.

These runtime savings will likely outweigh the deployment costs associated with the increased bytecode sizes.

Recommendations

As discussed, consider implementing private settings “getter” functions in an abstract contract that relevant contracts inherit from (at least for settings whose value is kept in `RocketStorage`).

Implementing this as an abstract contract still helps avoid code duplication, allowing the settings functions to be managed more centrally.

RP-30	RocketMinipool Deployment Gas Optimisations
Asset	`rocketpool: RocketMinipool.sol`
Status	Open
Rating	Informational

Description

`RocketMinipool` is a frequently deployed contract, with one deployed for every eth2 validator. As such, it is important to minimize deployment gas costs (which is the primary purpose of the proxy design that utilises `RocketMinipoolDelegate`). The testing team have identified some gas optimising modifications that may be worth considering.

These are primarily associated with reducing deployment costs, but some runtime gas optimisations are also included:

- Avoiding external calls in `OnlyRegisteredMinipool()` — if `RocketMinipool` contract instances are never upgraded (only the `RocketMinipoolDelegate` behind them), this modifier is only ever used to differentiate between the minipool proxy and the delegate contract.

  Provided this is the case, an effective check could just be to check the value of a local state variable like `isDelegate != true`. Another equivalent approach could be to expand the existing `status` variable such that the `RocketMinipoolDelegate` is constructed with an invalid status. `onlyMinipool()` could then look something like this:

  ```
  modifier onlyMinipool(address _minipoolAddress) {
      require(status != MinipoolStatus.Invalid, "Invalid minipool");
      _;
  }
  ```

- Reduce minipool bytecode size (and thus deployment costs) via moving construction/initialization code to `RocketMinipoolDelegate`.

  The majority of the logic contained in `RocketMinipool.constructor()` could be moved to a function in `RocketMinipoolDelegate` and executed as a `delegatecall` during construction.

  This would also remove the need to include `RocketNetworkFeesInterface` in `RocketMinipool` (though the majority of the associated bytecode is likely optimized away).

  This would also allow for the majority of the state variable definitions to be removed from `RocketMinipool` (as long as both contracts still define the `rocketStorage` variable at the same slot).

  NOTE: to prevent someone from having an unexpected impact by directly calling `RocketMinipoolDelegate.initialize()`, the delegate's constructor should ensure that the delegate does not have an "Initializable" status, the `initialize()` equivalent should only be callable when the contract has an "Initializable" status, *and* within `initialize()` the status should be set such that the contract is no longer "Initializable".[7]

- It's possible there are some bytecode savings introduced by importing an alternative to the `RocketStorageInterface` which only describes `getAddress()` (though this may be already optimized away).

[7] This is somewhat similar to the OpenZeppelin Beacon pattern

Recommendations

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

RP-31	Distributed Storage Key Namespace Design and Organisation
Asset	`rocketpool: contracts/*`
Status	Open
Rating	Informational

Description

The `RocketStorage` contract is used for centralized data storage, allowing other contracts to be replaced without migrating data. Values are stored in mappings keyed by bytes32, with separate mappings for each type.

The keys used are defined throughout the codebase, with some keys used across multiple contracts, others used across several but set only by one, and others used privately by a single contract. Although key prefixes like `"contract."` and `"rewards.pool.claim."` provide some context

There is some risk that maintainers accidentally reuse a key to store an unrelated value, resulting in a clash where storage values important to another contract are overwritten.

While no bugs have been identified in the current codebase associated with this issue, this can hinder long term maintainability — making it easy to accidentally introduce storage key clashes in unrelated contracts.

Consider the following example of a simple namespace clash, which can arise due to ambiguous encoding.

```
key_a = keccak256(abi.encodePacked("rocket.network.price",address(0)))
  key_b = keccak256(abi.encodePacked("rocket.network", uint64(3346300320100986928), address
        (0)))
  // where bytes8(3346300320100986928) == bytes8(".price00")
```

Here, the keys evaluate to the same storage slot, even though the prefixes are different.

See also the related issue RP-12.

Recommendations

Key prefixes/namespaces should be delimited from arguments via an otherwise unused character e.g. “.” This should also be used as a separator in `settingsNameSpace`.

It may be useful to distinguish between delimiters for sub-namespaces and the end of a key prefix (e.g. “.” is used within a key to separate hierarchical namespaces, but “!” is used to mark the end of a particular key, separating the key prefix from the `key argument` values).

Alternatively, you could enforce fixed key prefix sizes, such that the first 32 bytes of a keys pre-image are always the key prefix, and everything after is a “key argument”.

Consider implementing a centralized registry of storage key namespaces and associated contracts, optimally documenting or enforcing which protocol contracts are "responsible" for the storage keys (i.e. read vs write access). This can help identify bugs, should an unrelated contract start using the same key for something else.

Consider architecting a more formal namespace scheme, so it is more clear what keys are reserved for which contracts. For “privately controlled” values, one option could be to more rigorously mirror the path of the solidity files within the repo.

One feasible implementation could be to define all storage key prefixes and namespaces in a single abstract contract that all Rocket Pool contracts inherit from. Because Solidity constants are replaced at compile time[8], this should incur no runtime or deployment performance costs.

Consider checking that keys do not clash with any used elsewhere.

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

[8] See https://docs.soliditylang.org/en/v0.7.6/contracts.html#constant-and-immutable-state-variables

RP-32	Functions Can Be Declared External For Gas Savings
Asset	`rocketpool: contracts/*`
Status	Open
Rating	Informational

Description

The following functions can be declared external for some small gas savings:

- `RocketAuctionManager.getLotIsCleared()`
- `RocketMinipoolDelegate.getStatus()`
- `RocketMinipoolDelegate.getStatusBlock()`
- `RocketMinipoolDelegate.getStatusTime()`
- `RocketMinipoolDelegate.getDepositType()`
- `RocketMinipoolDelegate.getNodeAddress()`
- `RocketMinipoolDelegate.getNodeFee()`
- `RocketMinipoolDelegate.getNodeDepositBalance()`
- `RocketMinipoolDelegate.getNodeRefundBalance()`
- `RocketMinipoolDelegate.getNodeDepositAssigned()`
- `RocketMinipoolDelegate.getNodeWithdrawn()`
- `RocketMinipoolDelegate.getUserDepositBalance()`
- `RocketMinipoolDelegate.getUserDepositAssigned()`
- `RocketMinipoolDelegate.getUserDepositAssignedTime()`
- `RocketMinipoolDelegate.getStakingStartBalance()`
- `RocketMinipoolDelegate.getStakingEndBalance()`
- `RocketClaimDAO.getEnabled()`
- `RocketClaimDAO.spend(string,address,uint256)`
- `RocketClaimNode.getEnabled()`
- `RocketClaimNode.getClaimRewardsAmount(address)`

- `RocketClaimTrustedNode.getEnabled()`
- `RocketClaimTrustedNode.getClaimRewardsAmount(address)`
- `RocketDAONodeTrusted.getMemberQuorumVotesRequired()`
- `RocketDAONodeTrusted.getMemberIsValid(address)`
- `RocketDAONodeTrusted.getMemberAt(uint256)`
- `RocketDAONodeTrusted.getMemberMinRequired()`
- `RocketDAONodeTrusted.getMemberLastProposalBlock(address)`
- `RocketDAONodeTrusted.getMemberID(address)`
- `RocketDAONodeTrusted.getMemberEmail(address)`
- `RocketDAONodeTrusted.getMemberJoinedBlock(address)`
- `RocketDAONodeTrusted.getMemberProposalExecutedBlock(string,address)`
- `RocketDAONodeTrusted.getMemberRPLBondAmount(address)`
- `RocketDAONodeTrusted.getMemberReplacedAddress(string,address)`
- `RocketDAONodeTrusted.getMemberIsChallenged(address)`
- `RocketDAONodeTrusted.bootstrapMember(string,string,address)`
- `RocketDAONodeTrusted.bootstrapSettingUint(string,string,uint256)`
- `RocketDAONodeTrusted.bootstrapSettingBool(string,string,bool)`
- `RocketDAONodeTrusted.bootstrapUpgrade(string,string,string,address)`
- `RocketDAONodeTrusted.bootstrapDisable(bool)`
- `RocketDAONodeTrusted.memberJoinRequired(string,string)`
- `RocketDAONodeTrustedActions.actionJoin()`
- `RocketDAONodeTrustedActions.actionJoinRequired(address)`
- `RocketDAONodeTrustedProposals.propose(string,bytes)`
- `RocketDAONodeTrustedProposals.vote(uint256,bool)`
- `RocketDAONodeTrustedProposals.cancel(uint256)`
- `RocketDAONodeTrustedProposals.execute(uint256)`
- `RocketDAONodeTrustedProposals.proposalInvite(string,string,address)`
- `RocketDAONodeTrustedProposals.proposalLeave(address)`
- `RocketDAONodeTrustedProposals.proposalReplace(address,string,string,address)`

- `RocketDAONodeTrustedProposals.proposalKick(address,uint256)`
- `RocketDAONodeTrustedProposals.proposalSettingUint(string,string,uint256)`
- `RocketDAONodeTrustedProposals.proposalSettingBool(string,string,bool)`
- `RocketDAONodeTrustedProposals.proposalUpgrade(string,string,string,address)`
- `RocketDAONodeTrustedSettings.getSettingUint(string)`
- `RocketDAONodeTrustedSettings.setSettingUint(string,uint256)`
- `RocketDAONodeTrustedSettings.getSettingBool(string)`
- `RocketDAONodeTrustedSettings.setSettingBool(string,bool)`
- `RocketDAONodeTrustedSettingsMembers.getQuorum()`
- `RocketDAONodeTrustedSettingsMembers.getRPLBond()`
- `RocketDAONodeTrustedSettingsMembers.getMinipoolUnbondedMax()`
- `RocketDAONodeTrustedSettingsMembers.getChallengeCooldown()`
- `RocketDAONodeTrustedSettingsMembers.getChallengeWindow()`
- `RocketDAONodeTrustedSettingsMembers.getChallengeCost()`
- `RocketDAONodeTrustedSettingsProposals.getCooldown()`
- `RocketDAONodeTrustedSettingsProposals.getVoteBlocks()`
- `RocketDAONodeTrustedSettingsProposals.getVoteDelayBlocks()`
- `RocketDAONodeTrustedSettingsProposals.getExecuteBlocks()`
- `RocketDAONodeTrustedSettingsProposals.getActionBlocks()`
- `RocketDAONodeTrustedUpgrade.upgrade(string,string,string,address)`
- `RocketMinipoolManager.getMinipoolCount()`
- `RocketMinipoolManager.getMinipoolAt(uint256)`
- `RocketMinipoolManager.getNodeMinipoolCount(address)`
- `RocketMinipoolManager.getNodeMinipoolAt(address,uint256)`
- `RocketMinipoolManager.getNodeValidatingMinipoolCount(address)`
- `RocketMinipoolManager.getNodeValidatingMinipoolAt(address,uint256)`
- `RocketMinipoolManager.getMinipoolByPubkey(bytes)`
- `RocketMinipoolManager.getMinipoolExists(address)`
- `RocketMinipoolManager.getMinipoolPubkey(address)`

- `RocketMinipoolManager.getMinipoolWithdrawalTotalBalance(address)`
- `RocketMinipoolManager.getMinipoolWithdrawalNodeBalance(address)`
- `RocketMinipoolManager.getMinipoolWithdrawable(address)`
- `RocketMinipoolManager.getMinipoolWithdrawalProcessed(address)`
- `RocketMinipoolQueue.getTotalLength()`
- `RocketMinipoolQueue.getTotalCapacity()`
- `RocketMinipoolQueue.getEffectiveCapacity()`
- `RocketMinipoolQueue.getNextCapacity()`
- `RocketNetworkBalances.getTotalRETHSupply()`
- `RocketNetworkBalances.getETHUtilizationRate()`
- `RocketNetworkFees.getNodeFee()`
- `RocketNetworkPrices.getRPLPrice()`
- `RocketNodeManager.getNodeCount()`
- `RocketNodeManager.getNodeAt(uint256)`
- `RocketNodeManager.getNodeExists(address)`
- `RocketNodeManager.getNodeWithdrawalAddress(address)`
- `RocketNodeManager.getNodeTimezoneLocation(address)`
- `RocketNodeStaking.getTotalEffectiveRPLStake()`
- `RocketNodeStaking.getNodeEffectiveRPLStake(address)`
- `RocketNodeStaking.getNodeMinipoolLimit(address)`
- `RocketRewardsPool.getRPLBalance()`
- `RocketRewardsPool.getClaimBlockLastMade()`
- `RocketRewardsPool.getClaimingContractUserTotalCurrent(string)`
- `RocketTokenRETH.getExchangeRate()`
- `RocketTokenRETH.getCollateralRate()`
- `RocketTokenRPL.getInflationRewardsContractAddress()`
- `RocketTokenRPL.inflationMintTokens()`
- `RocketVault.balanceOf(string)`
- `RocketVault.balanceOfToken(string,address)`

- `RocketDAOProposal.getMessage(uint256)`
- `RocketDAOProposal.getCreated(uint256)`
- `RocketDAOProposal.getReceiptSupported(uint256,address)`
- `RocketDAOProposal.add(address,string,string,uint256,uint256,uint256,uint256,bytes)`
- `RocketDAOProposal.vote(address,uint256,uint256,bool)`
- `RocketDAOProposal.execute(uint256)`
- `RocketDAOProposal.cancel(address,uint256)`
- `RocketDAOProtocol.bootstrapSettingBool(string,string,bool)`
- `RocketDAOProtocol.bootstrapSettingAddress(string,string,address)`
- `RocketDAOProtocol.bootstrapSettingClaimer(string,uint256)`
- `RocketDAOProtocol.bootstrapSpendTreasury(string,address,uint256)`
- `RocketDAOProtocol.bootstrapDisable(bool)`
- `RocketDAOProtocolProposals.proposalSettingUint(string,string,uint256)`
- `RocketDAOProtocolProposals.proposalSettingBool(string,string,bool)`
- `RocketDAOProtocolProposals.proposalSettingAddress(string,string,address)`
- `RocketDAOProtocolProposals.proposalSettingRewardsClaimer(string,uint256)`
- `RocketDAOProtocolProposals.proposalSpendTreasury(string,address,uint256)`
- `RocketDAOProtocolSettings.getSettingAddress(string)`
- `RocketDAOProtocolSettings.setSettingAddress(string,address)`
- `RocketDAOProtocolSettingsAuction.getCreateLotEnabled()`
- `RocketDAOProtocolSettingsAuction.getBidOnLotEnabled()`
- `RocketDAOProtocolSettingsAuction.getLotMinimumEthValue()`
- `RocketDAOProtocolSettingsAuction.getLotMaximumEthValue()`
- `RocketDAOProtocolSettingsAuction.getLotDuration()`
- `RocketDAOProtocolSettingsAuction.getStartingPriceRatio()`
- `RocketDAOProtocolSettingsAuction.getReservePriceRatio()`
- `RocketDAOProtocolSettings.getSettingAddress(string)`
- `RocketDAOProtocolSettings.setSettingAddress(string,address)`
- `RocketDAOProtocolSettingsMinipool.getDepositNodeAmount(MinipoolDeposit)`

- `RocketDAOProtocolSettingsMinipool.getDepositUserAmount(MinipoolDeposit)`
- `RocketDAOProtocolSettingsMinipool.getSubmitWithdrawableEnabled()`
- `RocketDAOProtocolSettingsMinipool.getLaunchTimeout()`
- `RocketDAOProtocolSettingsMinipool.getWithdrawalDelay()`
- `RocketDAOProtocolSettingsRewards.getRewardsClaimerPercBlockUpdated(string)`

Recommendations

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

Consider including Slither or similar static analysis into the regular workflow or CI.

RP-33	Miscellaneous Rocket Pool Contract Issues
Asset	`rocketpool: contracts/*`
Status	Open
Rating	Informational

Description

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

- Unbonded validators have a RPL stake minimum that is 10% of 16 ether, not of 32 ether, so the staking requirement is effectively halved (even though there is no ETH collateral).

  Ensure this is intended.

- It is possible to send ETH directly to the minipool (e.g. via coinbase or selfdestruct) to bypass the type requirement.

  This can be used to stake before user funds have been provided by directly sending an extra 16 eth to a minipool of type "Half deposit". This is because `address(this).balance()` is used, rather than a state variable tracking correctly deposited funds.

  There were no associated security issues identified, but check that this is acceptable.

- `RocketTokenRPL:getInlfationIntervalsPassed()` should be corrected to `getInflationIntervalsPassed()` (and also in `RocketTokenRPLInterface.sol`).

- In `RocketRewardsPool.getClaimAmount()`, there is an unnecessary call to `getContractName(msg.sender)` at line [270].

  This value is already passed as an argument and is a bug, should `contractName != _claimingContract` hold.

- `RocketClaimNode.getEnabled()` should return false if the `RocketClaimNode` instance is not current i.e. when `getAddress("rocketClaimNode") != address(this)` holds.

- The following comments do not correctly match the functions they document.
  - At `RocketRewardsPool.getRPLBalance()` line [58], the comment for `@return` describes a `bool` but it should be a `uint256`.
  - At `RocketRewardsPool.getClaimingContractUserCanClaim()` line [163], the NatSpec documentation does not match the function.
  - At `RocketMinipoolDelegate.destroy()` line [346], the comment is inconsistent with the implementation. It describes “send[ing] any remaining ETH to [the] vault”, but it is actually sent to `nodeAddress`.
  - The comment for `RocketDaoNodeTrusted.bootstrapMember()` at line [161] states “if there are less than the required min amount of node members, the owner can add some to bootstrap the DAO”. However, the current code does not enforce anything with regards to a minimum required number of members.

    Update the comment, or don’t let the guardian invite using `bootstrapMember()` when the DAO reaches a certain member count.

- There are inconsistencies in the comment format throughout the contracts, some functions contain information about the return data, others just have a general comment, for example:
  - `RocketTokenNETH.sol` uses comment style `//`.
  - `RocketTokenRETH.sol` uses comment style `//`.
  - `RocketTokenRPL.sol` uses comment style `/** */`.
  - `RocketRewardsPool.sol` switches comment style from `/** */` and `//`.
- In `RocketMinipoolStatus` there are unnecessary external “getter” calls at lines [83,84], costing extra gas:

  ```
  minipool.getStakingStartBalance(),
        minipool.getStakingEndBalance()
  ```

  These values are already held in local variables.
- Revert message gas savings — consider replacing revert messages with comments (where not important for external interface) in order to reduce bytecode size and deployment costs.

  Some frameworks support “dev revert comments” that can be parsed and used to verify the correct revert was triggered in tests.[9]
- In `RocketDAONodeTrustedActions`, `ActionJoin` and `ActionJoinRequired` unnecessarily call `onlyRegisteredNode()` twice. This is also called in `_memberJoin()`.
- Gas saving optimisation in `RocketClaimNode.getClaimPossible()` — line [39]

  ```
  rocketNodeStaking.getNodeRPLStake(_nodeAddress) >= rocketNodeStaking.
         getNodeMinimumRPLStake(_nodeAddress)
  ```

  This can be replaced with a single external call to something like `rocketNodeStaking.isMinimumRPLStaked(_nodeAddress)`.

  In general, when there are multiple calls to the same contract using the same input, this should be straightforward to replace with a single external call.
- A similar gas saving optimisation in `RocketClaimNode.getClaimRewardsPerc()` — lines [51-53]

  ```
  uint256 totalRplStake = rocketNodeStaking.getTotalEffectiveRPLStake();
        if (totalRplStake == 0) { return 0; }
        return calcBase.mul(rocketNodeStaking.getNodeEffectiveRPLStake(\_nodeAddress)).div
          (totalRplStake);
  ```

  This code could be moved to `RocketNodeStaking` to minimize the number of external function calls.
- Gas saving optimisation for settings contracts — Because `settingNameSpace` is only defined during construction and never modified, there is no need for it to to be kept in a storage slot, costing extra gas to access. Instead, declare it as `immutable` [10]
- A typo in the comment at `RocketStorage.sol:36` — `guiardian` should be `guardian`.
- A typo in the comment at `RocketVault:75` — “it’s” should be “its”.
- `RocketNetworkPrices` uses an internally inconsistent namespace scheme for its storage keys. Some keys start with “network.price” and others “network.prices”.

[9] e.g. https://eth-brownie.readthedocs.io/en/stable/tests-pytest-intro.html#developer-revert-comments

[10] See https://docs.soliditylang.org/en/v0.7.6/contracts.html#immutable.

- Spelling mistake `_nodeChallengDeciderAddress` in `ActionChallengeDecided` event of `RocketDAONodeTrustedActions.sol`.
- `RocketDAOProposal.sol:execute()` has no access control and can be called by anyone. Not really an issue, but for consistency it should probably contain the `onlyDAOContract` modifier.
- Inefficient comparison of variables in `RocketDAONodeTrustedUpgrade.sol`. `typeHash` and `nameHash` can be compared against compile-time constants, instead of performing the hash each time.
- `LotCreated` event in `RocketAuctionManager.sol` uses `index` as one of its event arguments whereas other events refer to `index` as `lotIndex`.
- Proposals are not able to be voted on if it is the first block of the valid voting period. `RocketDAOProposal.sol:getState()` defines `ProposalState.Pending` as:

  ```
  block.number <= getStart(_proposalID)
  ```

  This can be updated to use a `<` sign instead of a `<=` sign, allowing DAO members to properly vote on proposals.
- Unused setting of `receipt.votes` in `RocketDAOProposal.sol`, taking up unecessary storage.
- Unclear to users that voting against a proposal in `RocketDAOProposal.sol` has no effect and abstaining from voting has the same effect. This should be made clear so that users don't unnecessarily vote against a proposal and waste gas on a pointless transaction.
- In `RocketNetworkPrices.submitPrices()` the same storage key prefix is used for 2 different types of keys. "network.prices.submitted.node" is used to construct keys containing different parameter sets at lines [[]56,61].

  While there is no danger of a collision in this case (as the keys consist of a different number of fixed-width parameters and no untrusted arguments of dynamic length), this can be unnecessarily confusing and sets an undesirable precedent.
- Potential DAO proposal gas savings: if the proposing account is a member of the relevant DAO, it likely makes sense for their proposal to count as a vote (to avoid having to make a separate vote transaction).

  Perhaps if it is unreasonable to expect that a proposal submission always indicates support, a boolean `voteInFavor` parameter could be added to the `propose()` function.
- It is unnecessary to initialise `RocketDaoNodeTrustedSettings.settingNameSpace` to `"` outside the constructor, as this is implied.
- In `RocketMinipoolDelegate` it should be reasonable to reduce storage costs by replacing `userDepositAssigned` with `userDepositBalance > 0`.
- Inconsistent spelling between `MinipoolStatus.Initialized` and the storage key "contract.storage.initialised". (Just nitpicking)
- Some contract names are missing from documentation e.g. "rocketTokenRPL" is not mentioned in `docs/contracts/design.html`.
- Inconsistent quotation in solidity code e.g. in `RocketRewardsPool` `'` is used at lines [62,64] but `"` is used elsewhere. Similarly in `RocketDaoNodeTrustedSettingsMembers`.

  Consider making use of a solidity code formatter e.g. prettier-plugin-solidity.

- Much of the contract code has very long lines e.g. `RocketRewardsPool` line [83] has 169 characters. This can be difficult to read on some systems or to identify relevant changes when viewing `git diff` results.

  Consider reformatting the contracts (e.g. with a code formatter like prettier-plugin-solidity) to reduce the maximum line length.

- The ABI interface for the Eth2 deposit contract used by Rocket Pool is incomplete or older, and is missing the `supportsInterface()` function.

  This constitutes no risk, but does mean that the function cannot be used, if desired.

- The Eth2 deposit contract bytecode used in unit testing is not the current bytecode used for the mainnet deposit contract.

  As this bytecode appears to be functionally equivalent (for the purposes of Rocket Pool testing) and is only used during testing, there is no identified security risk. However, there is some potential for the inconsistent bytecode to allow a unit test to pass that should fail.

  Consider updating the `casper/compiled/Deposit.bin` to match the mainnet contract.

- Slight `RocketVault` gas savings — `etherbalances` can be defined instead as `mapping(string => uint256)` to avoid an extra `keccak` operation.

- There is a warning when building the documentation:

  ```
  rocketpool/docs/smart-node/getting-started.rst:67: WARNING: Unexpected indentation.
  ```

  Add an empty line after the `::`.

Recommendations

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

Round Two Findings

The following entries describe issues identified within the Rocket Pool platform that were found during the second round of this assessment.

RP-34	Inaccurate RPL Inflation When Minting Multiple Intervals At Once		
Asset	`rocketpool: RocketTokenRPL.sol`		
Status	**Open**		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The inflationary RPL token is configured to mint additional tokens equivalent to 5% of its total supply annually (by default). This is implemented in `inflationMintTokens()`, which can mint additional tokens every `inflationInterval` (one day) at an equivalent rate. If `inflationMintTokens()` was not executed for some period, it is designed to mint additional tokens when next executed.

However, these calculations are not quite equivalent. Minting new tokens sporadically results in lower inflation than if they were minted every interval. This is illustrated in `test_can_get_same_from_multiple_intervals` [11] and is explained in more detail below.

This inaccuracy can result in reduced and unpredictable inflationary rewards, which could affect the validity of economic incentive analysis.

As a live RPL token contract cannot be upgraded, the likelihood of this issue having an effect is increased. Though, once identified, could be mitigated by ensuring `inflationMintTokens()` is executed every interval.

[11] – See Test Suite and the associated test implementation

Detailed Explanation

The inaccuracy occurs in `inflationCalculate()`, shown below:

```
    function inflationCalculate() override public view returns (uint256) {
126   // The inflation amount
      uint256 inflationTokenAmount = 0;
128   // Optimisation
      uint256 inflationRate = getInflationIntervalRate();
130   // Compute the number of inflation intervals elapsed since the last time we minted infation
          tokens
      uint256 intervalsSinceLastMint = getInflationIntervalsPassed();
132   // Only update if last interval has passed and inflation rate is > 0
      if(intervalsSinceLastMint > 0 && inflationRate > 0) {
134     // Our inflation rate
        uint256 rate = inflationRate;
136     // Compute inflation for total inflation intervals elapsed
        for (uint256 i = 1; i < intervalsSinceLastMint; i++) {
138       rate = rate.mul(inflationRate).div(10 ** 18);
        }
140     // Get the total supply now
        uint256 totalSupplyCurrent = totalSupply();
142     // Return inflation amount
        inflationTokenAmount = totalSupplyCurrent.mul(rate).div(10 ** 18).sub(totalSupplyCurrent)
            ;
144   }
      // Done
146   return inflationTokenAmount;
    }
```

When more than one interval has elapsed the loop at line [138] executes, intending to mimic the scenario where minting occurred every interval. This calculation would be equivalent should it occur in a system with arbitrarily accurate numbers and lossless division, but not in Solidity where division truncates to the integer.

When the loop executes, the truncating division is performed on the result of `rate.mul(inflationRate)` which is smaller than the result of `totalSupplyCurrent.mul(rate)` (because all values are positive, the inflation is increasing, and the total supply is much greater than `rate`).

In other words, the total supply after 2 intervals (minted each interval) is

```
totalSupplyCurrent.mul(inflationRate).div(10 ** 18).mul(inflationRate).div(10 ** 18)
```

but when minted in a single call to `inflationMintTokens()`, this would be

```
totalSupplyCurrent.mul(
  inflationRate.mul(inflationRate).div(10 ** 18)
).div(10 ** 18)
```

These are *not* equivalent in general.

Recommendations

Modify `inflationCalculate()` so that minting sporadically is equivalent to minting every interval.

One solution could involve something like

```
uint256 inflationRate = getInflationIntervalRate();
// Get the total supply now
uint256 totalSupplyCurrent = totalSupply();
uint256 newTotalSupply = totalSupplyCurrent;
// Compute inflation for total inflation intervals elapsed
for (uint256 i = 0; i < intervalsSinceLastMint; i++) {
  newTotalSupply = newTotalSupply.mul(inflationRate).div(10 ** 18);
}
// Return inflation amount
inflationTokenAmount = newTotalSupply.sub(totalSupplyCurrent);
```

Implement relevant tests to cover the scenario.

RP-35	Unexpected Behaviour If RPL Inflation Rate Set to Zero		
Asset	`rocketpool: RocketTokenRPL.sol`		
Status	**Open**		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The protocol DAO can configure the RPL inflation rate (which is initially set to the equivalent of 5% p.a. calculated daily). Should the inflation rate be set to zero for a period then back to some non-zero value, the behaviour of the minting calculations may be unexpected and undesirable.

While no tokens are minted when the inflation rate (`getInflationIntervalRate()`) is set to zero, the `inflationCalcTime` state variable is not updated so the contract acts equivalently to if `inflationMintTokens()` were never executed.

Consider the scenario where the inflation rate is set to zero for some period (of multiple one day intervals) P, and later set to a non-zero value at some time $T_{r>0}$. Intuitively, when executing `inflationMintTokens()` two intervals after $T_{r>0}$, we would expect to only mint tokens for those two intervals. Instead, tokens will be minted as if the rate had been non-zero for $P + 2$ intervals.

`inflationCalcTime` is only updated at line [164], after a requirement that number of tokens to mint is greater than zero.

Recommendations

Confirm whether the current behaviour is undesirable. If so, consider implementing a way to update the `inflationCalcTime` value when inflation intervals have passed while the inflation rate was set to zero.

RP-36	Node Operator Can Refuse to Distribute Minipool Funds		
Asset	`rocketpool: RocketMinipoolDelegate.sol`		
Status	Open		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

With recent changes to the Rocket Pool withdrawal process, a minipool's associated node operator is the only entity allowed to successfully execute `payout()`. This function is responsible for distributing the validator's funds to the node operator and the protocol (for use by rETH holders). If the node operator refuses to execute this function, these funds are effectively "locked" inside the minipool contract.

The likelihood of this being effectively abused is low. Because bonded minipools also usually contain a substantial amount of funds intended for the node operator, the operator has direct economic incentives to execute the function. This incentive can break down when the operator has "nothing to lose", like in the following, extreme cases:

- The Node Operator was grossly negligent or malicious such that the final balance was less than the `userDepositBalance` (less than 16 ETH for bonded minipools), so would receive no reward for calling payout.
- The minipool was unbonded so that the operator similarly has no funds locked in the minipool that would be released by `payout()`.

  As this could be reasonable grounds for "kicking" the operator from their Trusted Node DAO membership and slash their bond, this would be more relevant if the operator had *already* been kicked.

As any locked funds can be recovered via later `RocketMinipoolDelegate` contract updates, the operator cannot hold these funds hostage indefinitely and the impact is deemed low.

Recommendations

The testing team acknowledges the valid reason why access to `payout()` is restricted — to limit impact in the improbable event that the Trusted DAO incorrectly sets the minipool status to `Withdrawable` *before* the funds are delivered to the contract.

Consider allowing others to call the `payout()` in relevant circumstances. For example, when the minipool is unbonded or the `stakingEndBalance` is near or less than the `userDepositBalance`.

See also RP-37.

Mitigations could also involve the following:

- The "others" are still restricted e.g. to members of the Trusted DAO.
- Allow any Trusted DAO member to call `payout()` for unbonded minipools, so operators kicked from the DAO cannot misbehave.

- The “others” can only call an alternative version of `payout()`, which doesn’t `destroy()` the minipool. This is probably not beneficial unless it were also possible to process withdrawals more than once.

It may also be worth considering preventing withdrawal of staked RPL until the `payout` function has been executed for the associated minipool. As this would likely require additional book-keeping, the gas fees may be prohibitive (to stop counting the minipool as active for the purposes of receiving RPL rewards, but still active with regards to preventing withdrawal).

RP-37	Node Operator Can Revert `processWithdrawal()`		
Asset	`rocketpool: RocketNetworkWithdrawal.sol`		
Status	Open		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

This is related to RP-36. If `RocketMinipoolDelegate.payout()` were adjusted to allow accounts other than the node operator to call it, the operator could still make execution fail by reverting `RocketNetworkWithdrawal.processWithdrawal()`.

If the node is entitled to some of the validator balance, this is transferred to the node's withdrawal address via a low level call and must be successful (at line [60]).

```
// Transfer node ETH balance to node operator
if (nodeAmount > 0) {
  // Transfer ETH now
  (bool success,) = _nodeWithdrawalAddress.call{value: nodeAmount}("");
  require(success, "Node ETH balance was not successfully transferred to node withdrawal
        address");
}
```

The node operator can change the withdrawal address to a contract whose `receive()` or `fallback()` function is set to revert or consume all available gas, thus causing the transaction to fail.

Recommendations

If changing `RocketMinipoolDelegate.payout()` to allow execution by entities other than the node operator, ensure that the node operator cannot block execution by setting a malicious withdrawal address.

Mitigations could include:

- The alternative, exposed `payout()` does not deliver funds to the operator or destroy the minipool, only processing the user balance.

  The node operator would still need to call `payout()` to receive any allocated funds and destroy the contract.
- Instead of transferring the node funds to the withdrawal address via `RocketNetworkWithdrawal`, those funds are left in the minipool contract — to be delivered via `selfdestruct()`, which cannot fail.

  This has some drawbacks in requiring some refactoring of the `RocketNetworkWithdrawal` interface and limiting any processing logic possible in the node's withdrawal destination.

RP-38	Unhandled Errors — Round Two
Asset	`rocketpool-go` & `smartnode`
Status	Open
Rating	Informational

Description

A range of errors in go are not handled correctly. These errors have been deemed as informational, as no exploit over these errors could be identified.

Related code in `rocketpool-go`:

```go
// rocketpool-go/rocketpool/abi.go:28
  27:     }
> 28:     defer zlibReader.Close()
  29:
```

```go
// rocketpool-go/rocketpool/abi.go:48
  47:     zlibWriter := zlib.NewWriter(&abiCompressed)
> 48:     defer zlibWriter.Close()
  49:     if _, err := zlibWriter.Write([]byte(abiStr)); err != nil {
```

Related code `smartnode`

```go
// smartnode/rocketpool-cli/node/utils.go:36
  35:     if resp, err := http.Get(FreeGeoIPURL); err == nil {
> 36:       defer resp.Body.Close()
  37:       if body, err := ioutil.ReadAll(resp.Body); err == nil {
```

```go
// smartnode/rocketpool-cli/wallet/init.go:65
  64:     // Clear terminal output
> 65:     term.Clear()
  66:
```

```go
// smartnode/rocketpool-pow-proxy/proxy/http-proxy.go:62
  61:     log.Println(errors.New("Request Content-Type header not specified"))
> 62:     fmt.Fprintln(w, errors.New("Request Content-Type header not specified"))
  63:     return
```

```go
// smartnode/rocketpool-pow-proxy/proxy/http-proxy.go:70
  69:     log.Println(fmt.Errorf("Error forwarding request to remote server: %w", err))
> 70:     fmt.Fprintln(w, fmt.Errorf("Error forwarding request to remote server: %w", err))
  71:     return
```

```go
// smartnode/rocketpool-pow-proxy/proxy/http-proxy.go:73
  72:     }
> 73:     defer response.Body.Close()
  74:
```

```
// smartnode/rocketpool-pow-proxy/proxy/http-proxy.go:82
      log.Println(fmt.Errorf("Error reading response from remote server: %w", err))
>     fmt.Fprintln(w, fmt.Errorf("Error reading response from remote server: %w", err))
      return
```

```
// smartnode/rocketpool-pow-proxy/proxy/ws-proxy.go:62
      log.Println(fmt.Errorf("Error upgrading websocket: %w", err))
>     fmt.Fprintln(w, fmt.Errorf("Error upgrading websocket: %w", err))
      return
```

```
// smartnode/rocketpool-pow-proxy/proxy/ws-proxy.go:65
      }
>     defer eth2Connection.Close()

```

```
// smartnode/rocketpool-pow-proxy/proxy/ws-proxy.go:71
        log.Println(fmt.Errorf("Error connecting to remote websocket: %w", err))
>       fmt.Fprintln(w, fmt.Errorf("Error connecting to remote websocket: %w", err))
      }
```

```
// smartnode/rocketpool-pow-proxy/proxy/ws-proxy.go:73
      }
>     defer infuraConnection.Close()

```

```
// smartnode/rocketpool-pow-proxy/proxy/ws-proxy.go:86
      log.Println(fmt.Errorf("Error reading from eth2: %w", err))
>     fmt.Fprintln(w, fmt.Errorf("Error reading from eth2: %w", err))
      break
```

```
// smartnode/rocketpool-pow-proxy/proxy/ws-proxy.go:93
      log.Println(fmt.Errorf("Error writing to remote websocket: %w", err))
>     fmt.Fprintln(w, fmt.Errorf("Error writing to remote websocket: %w", err))
      break
```

```
// smartnode/rocketpool-pow-proxy/proxy/ws-proxy.go:108
      log.Println(fmt.Errorf("Error reading from remote websocket: %w", err))
>     fmt.Fprintln(w, fmt.Errorf("Error reading from remote websocket: %w", err))
      break
```

```
// smartnode/rocketpool-pow-proxy/proxy/ws-proxy.go:115
      log.Println(fmt.Errorf("Error writing to eth2: %w", err))
>     fmt.Fprintln(w, fmt.Errorf("Error writing to eth2: %w", err))
      break
```

```
// smartnode/rocketpool-pow-proxy/rocketpool-pow-proxy.go:83
      proxyServer := proxy.NewHttpProxyServer(c.GlobalString("httpPort"), c.GlobalString("httpProviderUrl"),
       c.GlobalString("network"), c.GlobalString("projectId"))
>     proxyServer.Start()
      wg.Done()
```

```
// smartnode/rocketpool-pow-proxy/rocketpool-pow-proxy.go:90
  89:     proxyServer := proxy.NewWsProxyServer(c.GlobalString("wsPort"), c.GlobalString("wsProviderUrl"), c.
         GlobalString("network"), c.GlobalString("projectId"))
> 90:     proxyServer.Start()
  91:     wg.Done()
```

```
// smartnode/shared/services/beacon/lighthouse/client.go:476
  475:     }
> 476:     defer response.Body.Close()
  477:
```

```
// smartnode/shared/services/beacon/lighthouse/client.go:505
  504:     }
> 505:     defer response.Body.Close()
  506:
```

```
// smartnode/shared/services/beacon/prysm/client.go:52
  51:     func (c *Client) Close() {
> 52:       c.conn.Close()
  53:     }
```

```
// smartnode/shared/services/beacon/teku/client.go:480
  479:     }
> 480:     defer response.Body.Close()
  481:
```

```
// smartnode/shared/services/beacon/teku/client.go:508
  507:     }
> 508:     defer response.Body.Close()
  509:
```

```
// smartnode/shared/services/rocketpool/client.go:146
  145:     if c.client != nil {
> 146:       c.client.Close()
  147:     }
```

```
// smartnode/shared/services/rocketpool/client.go:199
  198:     if err != nil { return err }
> 199:     defer cmd.Close()
  200:
```

```
// smartnode/shared/services/rocketpool/client.go:223
  222:     if verbose {
> 223:       c.Println(scanner.Text())
  224:     }
```

```
// smartnode/shared/services/rocketpool/client.go:517
  516:     if err != nil { return err }
> 517:     defer cmd.Close()
  518:
```

```
// smartnode/shared/services/rocketpool/client.go:524
            if err != nil { return err }
>           go io.Copy(os.Stdout, cmdOut)
            go io.Copy(os.Stderr, cmdErr)
```

```
// smartnode/shared/services/rocketpool/client.go:525
            go io.Copy(os.Stdout, cmdOut)
>           go io.Copy(os.Stderr, cmdErr)

```

```
// smartnode/shared/services/rocketpool/client.go:541
            }
>           defer cmd.Close()

```

Recommendations

Consider propagating or handling the errors for each of these cases.

Consider integrating errcheck and other security linters into the CI workflow.

Additionally, avoid using the `defer` keyword to close writable files as the `Close()` method returns an error value. Although a common idiom, by using the `defer` keyword, the returned error value is being ignored. While uncommon, an error returned by `Close()` could indicate a problem with the filesystem and the writes were not successfully flushed to disk.

RP-39	Frequent RPL Reward Claim Requirement Unevenly Impacts Small Node Operators
Asset	`rocketpool: RocketClaimNode.sol`
Status	Open
Rating	Informational

Description

Rocket Pool defines 14 days as the default period at which staking participants are required too claim any fees generated through RPL rewards. This RPL acts as insurance against poor node operator performance and simultaneously protects user's deposits. The rewarded RPL is determined by a node operator's share of the entire staked pool of RPL.

For a node operator who only runs one minipool and stakes the minimum required RPL (10%), it's possible that the gas fees required to claim RPL every two weeks could significantly limit their profitability in comparison to a node operator who runs several Rocket Pool minipools. This could be mitigated by tracking the profitability of a node operator from when it begins staking and when it decides to claim RPL after withdrawing their ETH. Without reducing the number of times a node operator is required to claim RPL to some fixed amount, the Rocket Pool system is potentially threatened by more centralized node operators who are able to afford these costs.

Recommendations

The testing team recommends Rocket Pool re-implements the distribution of inflation rewards to each of the claiming contracts using an indexation mechanism. Each claiming contract will then have an index which tracks the total reward amounts, updated at the start of each claiming period. The index of each node operator will be tracked upon entering and exiting the RPL staking system. Changes to the distribution of inflation as in `RocketDaoProtocolSettingsRewards`, should not affect any indexation calculations.

There are a few drawbacks to the proposed implementation:

- Increased gas costs when minting RPL. Upon each inflation period, the index value of each registered claiming contract will need to be updated.
- Additional claimers would increase the amount of gas required to iterate through the loop. This can be mitigated by limiting the maximum number of registered claiming contracts allowed.
- Added contract complexity due to the implementation of performing index calculations and updates.
- Increased gas costs for the end user when staking, withdrawing and updating their stake.
- It may be infeasible to track consequences when a node operator occasionally goes above 150% or below 10% per minipool.

However, despite these drawbacks, indexation provides increased flexibility to smaller node operators in a way that improves overall user experience. It's important to note that if the current implementation is left untouched, that node operators understand and accept the running costs of claiming RPL every two weeks.

An alternative mitigation could include weighting the distribution of RPL inflation rewards more heavily to small stakers, to counteract the proportionally larger gas fees.

RP-40	Inconsistent User Deposit Gas Estimation
Asset	`rocketpool: RocketDepositPool.sol`
Status	Open
Rating	Informational

Description

The amount of gas consumed by the user-facing `RocketDepositPool.deposit()` function can fluctuate reasonably significantly, potentially causing problems for UIs trying to estimate a reasonable maximum gas limit.

The gas consumed depends depends on the `RocketDepositPool` s ETH balance when the `deposit()` transaction is mined. As an example, a deposit for 1 ETH would not execute `RocketMinipoolDelegate.userDeposit()` when the deposit pool has a balance of 13 ETH (not enough to fund any minipool), but would execute it twice when the pool has a balance of 100 ETH.[12] This balance can vary widely within a single block based on deposit ordering, burnt rETH, and processed validator withdrawals.

If a user's UI (web UI or wallet) relies on the `eth_estimateGas` JSONRPC call to suggest a maximum gas limit for their transaction it may return a value assuming no deposit assignments occur but, when the transaction is mined, it must perform these assignments. Thus the transaction fails to execute due to an insufficient gas.

This has no identified security impact, but could be a UI "pain point".

Recommendations

Have the UI set or recommend a default `gasLimit` that is sufficient to execute `rocketDAOProtocolSettingsDeposit.getMaximumDepositAssignments()` deposit assignments.

This could involve:

- `RocketDepositPool.deposit()` containing a `require(gas >= minRecommendedGas)` statement.

  This would ensure `eth_estimateGas()` returns a suitable value to appease this, but unnecessarily sets a hard requirement in the contract which could be troublesome to maintain.
- Have the official UI specify a `gasLimit` that does not use `eth_estimateGas()` alone to spontaneously calculate the value, instead returning a value sufficient to execute the maximum possible number of deposits giventhe current settings.

The primary complications involved are likely associated with updating any estimate when protocol settings or contract code changes.

Consider also documenting this (and a suitable gasLimit) in relevant user documentation, to help users who want to submit funds directly from their wallet application. It might be helpful to maintain a test network that mirrors the contract code and settings of mainnet.

[12] Assuming deposit assignment is enabled (`RocketDAOProtocolSettingsDeposit.getAssignDepositsEnabled()`) and other settings remain their default values.

RP-41	Miscellaneous Rocket Pool Contract Issues — Round Two
Asset	`rocketpool: contracts/*`
Status	Open
Rating	Informational

Description

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

- The contract compilation process does not include an optimisation step. That is, the project's `truffle-config.js` does not contain `optimizer: { enabled: true }`.

  These various optimisations can improve gas efficiency and reduce bytecode size.
- The nETH token was removed, but several references remain in the documentation and supporting files. Specifically, in the following files:
  - `docs/rocket-pool/reward-tokens.rst`
  - `docs/rocket-pool/minipools.rst`
  - `docs/rocket-pool/staking.rst`
  - `docs/rocket-pool/nodes.rst`
  - `docs/js-library/tokens.rst`
  - `docs/js-library/getting-started.rst`
  - `docs/js-library/settings.rst`
  - `docs/contracts/design.rst`
  - `docs/smart-node/minipools.rst`
  - `docs/smart-node/node-setup.rst`
  - `test/_helpers/tokens.js`
  - `test/minipool/scenario-close.js`
  - `test/minipool/scenario-withdraw.js`
- Gas saving optimisation in `rocketMinipoolQueue.removeMinipool()` — An external contract call can be avoided (at line [126]) by taking the deposit type as a function parameter.

  Because `removeMinipool()` can only be called by a registered minipool, this requires no additional trust assumption.
- Gas saving optimisation in `RocketMinipoolDelegate.setWithdrawable()` — lines [178–179] (shown below)

```
RocketMinipoolQueueInterface rocketMinipoolQueue = RocketMinipoolQueueInterface(
        getContractAddress("rocketMinipoolQueue"));
if (!userDepositAssigned) { rocketMinipoolQueue.removeMinipool(); }
```

can be rewritten as

```
if (!userDepositAssigned) {
  RocketMinipoolQueueInterface rocketMinipoolQueue = RocketMinipoolQueueInterface(
        getContractAddress("rocketMinipoolQueue"));
  rocketMinipoolQueue.removeMinipool();
}
```

to avoid an extra external call in the majority of cases.

This is a minor improvement, as is executed comparatively infrequently.

- The comment `RocketTokenRPL.sol:30` is inaccurate — "Last block inflation was calculated at" should be modified to something like "Timestamp of last block inflation was calculated at" since the timestamp is now tracked instead of the block number.

- Slight typo at `RocketTokenRPL.sol:130` — "infation" should be "inflation".

- `RocketDepositPool.assignDeposits()` could be refactored in a more gas efficient manner to only make a single call to `getBalance()` and `rocketVault.withdrawEther()` (by tracking balance changes in the local contract and using two loops).

- `RocketDepositPool.assignDeposits()` could reduce the number of external `getContractAddress("rocketVault")` function calls, by calling a private version of `getBalance()` that accepts the `RocketVaultInterface` as a parameter.

- PR #201 introduces some additional checks (in `_beforeTokenTransfer()`) before performing rETH transfers or burns. Some gas could be saved by reducing the number of calls to external contracts.

  By directly calling `getUintS("dao.protocol.setting.networknetwork.reth.deposit.delay")`, we can avoid an external call to storage for `getContractAddress("rocketDAOProtocolSettingsNetwork")` and one of the external calls involved with executing `rocketDAOProtocolSettingsNetwork.getRethDepositDelay()` .

  Although this has some limits on upgradability (the value can change but not the storage key), this would already be restricted to never changing the `getRethDepositDelay()` function signature.

- Like in RP-31 the testing team recommends that the storage key prefix `"user.deposit.block"` (defined in PR #201) were instead delimited like `"user.deposit.block."` or `"user.deposit.block!"`

- Given that PR #201 prevents users from withdrawing or transferring their rETH for some period after any deposit, ensure user documentation clearly explains and describes this behaviour.

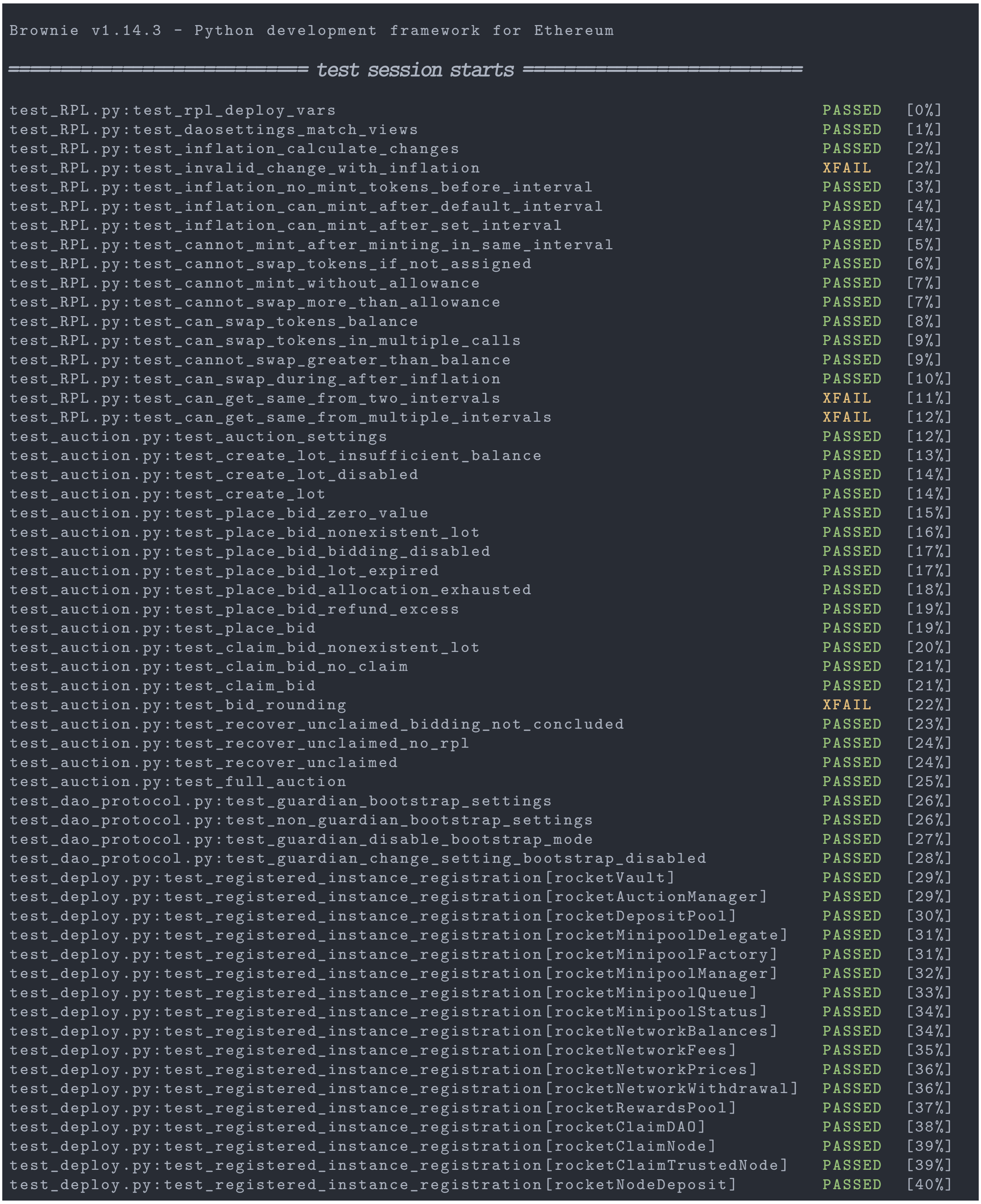
  This should also, optimally, have a relevant notice or warning in the official UI.

Recommendations

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

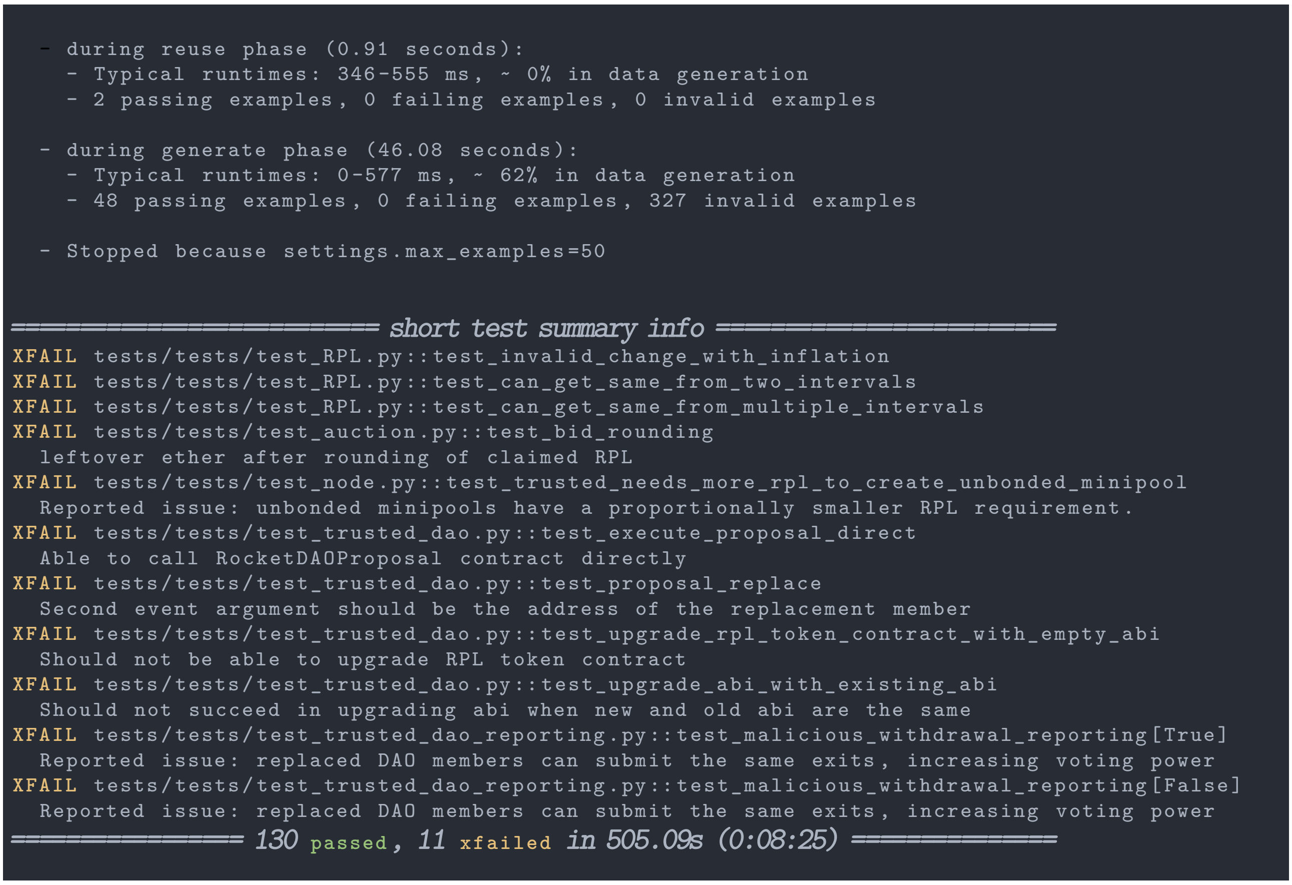
Appendix A Test Suite

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



```
Brownie v1.14.3 - Python development framework for Ethereum

============================= test session starts ===========================

test_RPL.py:test_rpl_deploy_vars                                                PASSED  [0%]
test_RPL.py:test_daosettings_match_views                                        PASSED  [1%]
test_RPL.py:test_inflation_calculate_changes                                    PASSED  [2%]
test_RPL.py:test_invalid_change_with_inflation                                  XFAIL   [2%]
test_RPL.py:test_inflation_no_mint_tokens_before_interval                       PASSED  [3%]
test_RPL.py:test_inflation_can_mint_after_default_interval                      PASSED  [4%]
test_RPL.py:test_inflation_can_mint_after_set_interval                          PASSED  [4%]
test_RPL.py:test_cannot_mint_after_minting_in_same_interval                     PASSED  [5%]
test_RPL.py:test_cannot_swap_tokens_if_not_assigned                             PASSED  [6%]
test_RPL.py:test_cannot_mint_without_allowance                                  PASSED  [7%]
test_RPL.py:test_cannot_swap_more_than_allowance                                PASSED  [7%]
test_RPL.py:test_can_swap_tokens_balance                                        PASSED  [8%]
test_RPL.py:test_can_swap_tokens_in_multiple_calls                              PASSED  [9%]
test_RPL.py:test_cannot_swap_greater_than_balance                               PASSED  [9%]
test_RPL.py:test_can_swap_during_after_inflation                                PASSED  [10%]
test_RPL.py:test_can_get_same_from_two_intervals                                XFAIL   [11%]
test_RPL.py:test_can_get_same_from_multiple_intervals                           XFAIL   [12%]
test_auction.py:test_auction_settings                                           PASSED  [12%]
test_auction.py:test_create_lot_insufficient_balance                            PASSED  [13%]
test_auction.py:test_create_lot_disabled                                        PASSED  [14%]
test_auction.py:test_create_lot                                                 PASSED  [14%]
test_auction.py:test_place_bid_zero_value                                       PASSED  [15%]
test_auction.py:test_place_bid_nonexistent_lot                                  PASSED  [16%]
test_auction.py:test_place_bid_bidding_disabled                                 PASSED  [17%]
test_auction.py:test_place_bid_lot_expired                                      PASSED  [17%]
test_auction.py:test_place_bid_allocation_exhausted                             PASSED  [18%]
test_auction.py:test_place_bid_refund_excess                                    PASSED  [19%]
test_auction.py:test_place_bid                                                  PASSED  [19%]
test_auction.py:test_claim_bid_nonexistent_lot                                  PASSED  [20%]
test_auction.py:test_claim_bid_no_claim                                         PASSED  [21%]
test_auction.py:test_claim_bid                                                  PASSED  [21%]
test_auction.py:test_bid_rounding                                               XFAIL   [22%]
test_auction.py:test_recover_unclaimed_bidding_not_concluded                    PASSED  [23%]
test_auction.py:test_recover_unclaimed_no_rpl                                   PASSED  [24%]
test_auction.py:test_recover_unclaimed                                          PASSED  [24%]
test_auction.py:test_full_auction                                               PASSED  [25%]
test_dao_protocol.py:test_guardian_bootstrap_settings                           PASSED  [26%]
test_dao_protocol.py:test_non_guardian_bootstrap_settings                       PASSED  [26%]
test_dao_protocol.py:test_guardian_disable_bootstrap_mode                       PASSED  [27%]
test_dao_protocol.py:test_guardian_change_setting_bootstrap_disabled            PASSED  [28%]
test_deploy.py:test_registered_instance_registration[rocketVault]               PASSED  [29%]
test_deploy.py:test_registered_instance_registration[rocketAuctionManager]      PASSED  [29%]
test_deploy.py:test_registered_instance_registration[rocketDepositPool]         PASSED  [30%]
test_deploy.py:test_registered_instance_registration[rocketMinipoolDelegate]    PASSED  [31%]
test_deploy.py:test_registered_instance_registration[rocketMinipoolFactory]     PASSED  [31%]
test_deploy.py:test_registered_instance_registration[rocketMinipoolManager]     PASSED  [32%]
test_deploy.py:test_registered_instance_registration[rocketMinipoolQueue]       PASSED  [33%]
test_deploy.py:test_registered_instance_registration[rocketMinipoolStatus]      PASSED  [34%]
test_deploy.py:test_registered_instance_registration[rocketNetworkBalances]     PASSED  [34%]
test_deploy.py:test_registered_instance_registration[rocketNetworkFees]         PASSED  [35%]
test_deploy.py:test_registered_instance_registration[rocketNetworkPrices]       PASSED  [36%]
test_deploy.py:test_registered_instance_registration[rocketNetworkWithdrawal]   PASSED  [36%]
test_deploy.py:test_registered_instance_registration[rocketRewardsPool]         PASSED  [37%]
test_deploy.py:test_registered_instance_registration[rocketClaimDAO]            PASSED  [38%]
test_deploy.py:test_registered_instance_registration[rocketClaimNode]           PASSED  [39%]
test_deploy.py:test_registered_instance_registration[rocketClaimTrustedNode]    PASSED  [39%]
test_deploy.py:test_registered_instance_registration[rocketNodeDeposit]         PASSED  [40%]
test_deploy.py:test_registered_instance_registration[rocketNodeManager]        PASSED  [41%]
test_deploy.py:test_registered_instance_registration[rocketNodeStaking]        PASSED  [41%]
test_deploy.py:test_registered_instance_registration[rocketDAOProposal]        PASSED  [42%]
test_deploy.py:test_registered_instance_registration[rocketDAONodeTrusted]     PASSED  [43%]
test_deploy.py:test_registered_instance_registration                           PASSED  [43%]
    [rocketDAONodeTrustedProposals]
test_deploy.py:test_registered_instance_registration                           PASSED  [44%]
    [rocketDAONodeTrustedActions]
test_deploy.py:test_registered_instance_registration                           PASSED  [45%]
    [rocketDAONodeTrustedUpgrade]
test_deploy.py:test_registered_instance_registration                           PASSED  [46%]
    [rocketDAONodeTrustedSettingsMembers]
test_deploy.py:test_registered_instance_registration                           PASSED  [46%]
    [rocketDAONodeTrustedSettingsProposals]
test_deploy.py:test_registered_instance_registration[rocketDAOProtocol]        PASSED  [47%]
test_deploy.py:test_registered_instance_registration                           PASSED  [48%]
    [rocketDAOProtocolProposals]
test_deploy.py:test_registered_instance_registration                           PASSED  [48%]
    [rocketDAOProtocolActions]
test_deploy.py:test_registered_instance_registration                           PASSED  [49%]
    [rocketDAOProtocolSettingsInflation]
test_deploy.py:test_registered_instance_registration                           PASSED  [50%]
    [rocketDAOProtocolSettingsRewards]
test_deploy.py:test_registered_instance_registration                           PASSED  [51%]
    [rocketDAOProtocolSettingsAuction]
test_deploy.py:test_registered_instance_registration                           PASSED  [51%]
    [rocketDAOProtocolSettingsNode]
test_deploy.py:test_registered_instance_registration                           PASSED  [52%]
    [rocketDAOProtocolSettingsNetwork]
test_deploy.py:test_registered_instance_registration                           PASSED  [53%]
    [rocketDAOProtocolSettingsDeposit]
test_deploy.py:test_registered_instance_registration                           PASSED  [53%]
    [rocketDAOProtocolSettingsMinipool]
test_deploy.py:test_registered_instance_registration                           PASSED  [54%]
    [rocketTokenRPLFixedSupply]
test_deploy.py:test_registered_instance_registration[rocketTokenRETH]          PASSED  [55%]
test_deploy.py:test_registered_instance_registration[rocketTokenNETH]          PASSED  [56%]
test_deploy.py:test_registered_instance_registration[rocketTokenRPL]           PASSED  [56%]
test_deploy.py:test_registered_instance_registration[addressQueueStorage]      PASSED  [57%]
test_deploy.py:test_registered_instance_registration[addressSetStorage]        PASSED  [58%]
test_deploy.py:test_registered_instance_registration[casperDeposit]            PASSED  [58%]
test_deploy.py:test_registered_abi_only                                        PASSED  [59%]
test_deploy.py:test_storage_should_be_initialised                              PASSED  [60%]
test_deploy.py:test_owner_should_not_be_registered_as_a_contract               PASSED  [60%]
test_minipool_reward.py:test_get_minipool_reward_positive[0]                   PASSED  [61%]
test_minipool_reward.py:test_get_minipool_reward_positive[16 ether]            PASSED  [62%]
test_minipool_reward.py:test_get_minipool_reward_positive[32 ether]            PASSED  [63%]
test_minipool_reward.py:test_get_minipool_reward_negative[0]                   PASSED  [63%]
test_minipool_reward.py:test_get_minipool_reward_negative[16 ether]            PASSED  [64%]
test_minipool_reward.py:test_get_minipool_reward_negative[32 ether]            PASSED  [65%]
test_nETH.py:test_simple_deploy_vars                                           PASSED  [65%]
test_nETH.py:test_zero_address_constructor                                     PASSED  [66%]
test_nETH.py:test_only_network_can_deposit                                     PASSED  [67%]
test_nETH.py:test_deposit_reward_respects_updates                              PASSED  [68%]
test_nETH.py:test_only_minipool_can_mint                                       PASSED  [68%]
test_nETH.py:test_can_correctly_burn                                           PASSED  [69%]
test_nETH.py:test_cannot_burn_zero_balance                                     PASSED  [70%]
test_nETH.py:test_cannot_burn_zero_eth_contract                                PASSED  [70%]
test_nETH.py:test_cannot_burn_more_than_eth                                    PASSED  [71%]
test_node.py:test_cannot_create_minipool_without_rpl_stake                     PASSED  [72%]
test_node.py:test_can_create_minipool_when_staking_rpl                         PASSED  [73%]
    [MinipoolDepositCls.FULL]
test_node.py:test_can_create_minipool_when_staking_rpl                         PASSED  [73%]
    [MinipoolDepositCls.HALF]
test_node.py:test_normal_node_cannot_create_unbonded_minipool                  PASSED  [74%]
test_node.py:test_trusted_node_cannot_create_without_staked_rpl                PASSED  [75%]
    [MinipoolDepositCls.FULL]
test_node.py:test_trusted_can_create_minipool_with_any_deposit_amount          PASSED  [75%]
    [MinipoolDepositCls.FULL]
test_node.py:test_trusted_node_cannot_create_without_staked_rpl                PASSED  [76%]
```

```
    [MinipoolDepositCls.HALF]
test_node.py:test_trusted_can_create_minipool_with_any_deposit_amount          PASSED   [77%]
    [MinipoolDepositCls.HALF]
test_node.py:test_trusted_node_cannot_create_without_staked_rpl                PASSED   [78%]
    [MinipoolDepositCls.EMPTY]
test_node.py:test_trusted_can_create_minipool_with_any_deposit_amount          PASSED   [78%]
    [MinipoolDepositCls.EMPTY]
test_node.py:test_trusted_needs_more_rpl_to_create_unbonded_minipool           XFAIL    [79%]
test_rETH.py:test_rETH_deployment_params                                       PASSED   [80%]
test_rETH.py:test_rETH_zero_storage                                            PASSED   [80%]
test_rETH.py:test_only_network_can_deposit                                     PASSED   [81%]
test_rETH.py:test_deposit_reward_respects_updates                              PASSED   [82%]
test_rETH.py:test_can_get_correct_eth_value_at_start                           PASSED   [82%]
test_storage.py:test_can_retrieve_some_stored_uint                             PASSED   [83%]
test_storage.py:test_can_retrieve_some_stored_string                           PASSED   [84%]
test_storage.py:test_can_retrieve_some_stored_address                          PASSED   [85%]
test_storage.py:test_can_retrieve_some_stored_bytes                            PASSED   [85%]
test_storage.py:test_can_retrieve_some_stored_bool                             PASSED   [86%]
test_storage.py:test_can_retrieve_some_stored_int                              PASSED   [87%]
test_storage.py:test_can_retrieve_some_stored_bytes32                          PASSED   [87%]
test_storage.py:test_different_types_are_independent                           PASSED   [88%]
test_storage.py:test_locking                                                   PASSED   [89%]
test_trusted_dao.py:test_create_proposal_and_execute                           PASSED   [90%]
test_trusted_dao.py:test_execute_proposal_empty_calldata                       PASSED   [90%]
test_trusted_dao.py:test_cancel_proposal                                       PASSED   [91%]
test_trusted_dao.py:test_execute_proposal_direct                               XFAIL    [92%]
test_trusted_dao.py:test_action_join_no_invite                                 PASSED   [92%]
test_trusted_dao.py:test_action_join_required                                  PASSED   [93%]
test_trusted_dao.py:test_proposal_leave                                        PASSED   [94%]
test_trusted_dao.py:test_proposal_replace                                      XFAIL    [95%]
test_trusted_dao.py:test_proposal_kick                                         PASSED   [95%]
test_trusted_dao.py:test_challenge_node                                        PASSED   [96%]
test_trusted_dao.py:test_upgrade_rpl_token_contract_with_empty_abi             XFAIL    [97%]
test_trusted_dao.py:test_upgrade_abi_with_existing_abi                         XFAIL    [97%]
test_trusted_dao_reporting.py:test_correct_withdrawal                          PASSED   [98%]
test_trusted_dao_reporting.py:test_malicious_withdrawal_reporting[True]        XFAIL    [99%]
test_trusted_dao_reporting.py:test_malicious_withdrawal_reporting[False]       XFAIL    [100%]

============================ Hypothesis Statistics ============================

test_minipool_reward.py:test_get_minipool_reward_positive[0]:

  - during reuse phase (0.04 seconds):
    - Typical runtimes: ~ 15ms, ~ 2% in data generation
    - 1 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (10.28 seconds):
    - Typical runtimes: 0-216 ms, ~ 23% in data generation
    - 49 passing examples, 0 failing examples, 17 invalid examples

  - Stopped because settings.max_examples=50


test_minipool_reward.py:test_get_minipool_reward_positive[16 ether]:

  - during reuse phase (0.24 seconds):
    - Typical runtimes: ~ 217ms, ~ 0% in data generation
    - 1 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (10.31 seconds):
    - Typical runtimes: 0-215 ms, ~ 23% in data generation
    - 49 passing examples, 0 failing examples, 17 invalid examples

  - Stopped because settings.max_examples=50


test_minipool_reward.py:test_get_minipool_reward_positive[32 ether]:

  - during reuse phase (0.25 seconds):
    - Typical runtimes: ~ 222ms, ~ 0% in data generation
    - 1 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (10.22 seconds):
    - Typical runtimes: 0-215 ms, ~ 25% in data generation
    - 49 passing examples, 0 failing examples, 18 invalid examples

  - Stopped because settings.max_examples=50


test_minipool_reward.py:test_get_minipool_reward_negative[0]:

  - during reuse phase (0.25 seconds):
    - Typical runtimes: ~ 223ms, ~ 1% in data generation
    - 1 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (10.40 seconds):
    - Typical runtimes: 0-229 ms, ~ 23% in data generation
    - 49 passing examples, 0 failing examples, 17 invalid examples

  - Stopped because settings.max_examples=50


test_minipool_reward.py:test_get_minipool_reward_negative[16 ether]:

  - during reuse phase (0.32 seconds):
    - Typical runtimes: ~ 248ms, ~ 0% in data generation
    - 1 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (10.33 seconds):
    - Typical runtimes: 0-218 ms, ~ 24% in data generation
    - 49 passing examples, 0 failing examples, 17 invalid examples

  - Stopped because settings.max_examples=50


test_minipool_reward.py:test_get_minipool_reward_negative[32 ether]:

  - during reuse phase (0.23 seconds):
    - Typical runtimes: ~ 206ms, ~ 0% in data generation
    - 1 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (10.20 seconds):
    - Typical runtimes: 0-215 ms, ~ 24% in data generation
    - 49 passing examples, 0 failing examples, 17 invalid examples

  - Stopped because settings.max_examples=50


test_storage.py:test_can_retrieve_some_stored_uint:

  - during reuse phase (0.06 seconds):
    - Typical runtimes: ~ 57ms, ~ 0% in data generation
    - 1 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (2.36 seconds):
    - Typical runtimes: 258-266 ms, ~ 0% in data generation
    - 9 passing examples, 0 failing examples, 0 invalid examples

  - Stopped because settings.max_examples=10


test_storage.py:test_can_retrieve_some_stored_string:

  - during reuse phase (0.38 seconds):
    - Typical runtimes: 109-266 ms, ~ 0% in data generation
    - 2 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (2.14 seconds):
    - Typical runtimes: 0-286 ms, ~ 32% in data generation
    - 8 passing examples, 0 failing examples, 4 invalid examples
  - Stopped because settings.max_examples=10


test_storage.py:test_can_retrieve_some_stored_address:

  - during reuse phase (0.07 seconds):
    - Typical runtimes: ~ 64ms, ~ 0% in data generation
    - 1 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (2.41 seconds):
    - Typical runtimes: 261-296 ms, ~ 0% in data generation
    - 9 passing examples, 0 failing examples, 0 invalid examples

  - Stopped because settings.max_examples=10


test_storage.py:test_can_retrieve_some_stored_bytes:

  - during reuse phase (0.09 seconds):
    - Typical runtimes: ~ 66ms, ~ 1% in data generation
    - 1 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (2.45 seconds):
    - Typical runtimes: 260-297 ms, ~ 0% in data generation
    - 9 passing examples, 0 failing examples, 0 invalid examples

  - Stopped because settings.max_examples=10


test_storage.py:test_can_retrieve_some_stored_bool:

  - during reuse phase (0.16 seconds):
    - Typical runtimes: ~ 129ms, ~ 0% in data generation
    - 1 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (2.53 seconds):
    - Typical runtimes: 264-336 ms, ~ 0% in data generation
    - 9 passing examples, 0 failing examples, 0 invalid examples

  - Stopped because settings.max_examples=10


test_storage.py:test_can_retrieve_some_stored_int:

  - during reuse phase (0.09 seconds):
    - Typical runtimes: ~ 63ms, ~ 1% in data generation
    - 1 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (2.54 seconds):
    - Typical runtimes: 259-368 ms, ~ 0% in data generation
    - 9 passing examples, 0 failing examples, 0 invalid examples

  - Stopped because settings.max_examples=10


test_storage.py:test_can_retrieve_some_stored_bytes32:

  - during reuse phase (0.09 seconds):
    - Typical runtimes: ~ 67ms, ~ 0% in data generation
    - 1 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (2.45 seconds):
    - Typical runtimes: 258-294 ms, ~ 0% in data generation
    - 9 passing examples, 0 failing examples, 0 invalid examples

  - Stopped because settings.max_examples=10


test_storage.py:test_different_types_are_independent:
```



```
  - during reuse phase (0.91 seconds):
    - Typical runtimes: 346-555 ms, ~ 0% in data generation
    - 2 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (46.08 seconds):
    - Typical runtimes: 0-577 ms, ~ 62% in data generation
    - 48 passing examples, 0 failing examples, 327 invalid examples

  - Stopped because settings.max_examples=50


=========================== short test summary info ===========================
XFAIL tests/tests/test_RPL.py::test_invalid_change_with_inflation
XFAIL tests/tests/test_RPL.py::test_can_get_same_from_two_intervals
XFAIL tests/tests/test_RPL.py::test_can_get_same_from_multiple_intervals
XFAIL tests/tests/test_auction.py::test_bid_rounding
  leftover ether after rounding of claimed RPL
XFAIL tests/tests/test_node.py::test_trusted_needs_more_rpl_to_create_unbonded_minipool
  Reported issue: unbonded minipools have a proportionally smaller RPL requirement.
XFAIL tests/tests/test_trusted_dao.py::test_execute_proposal_direct
  Able to call RocketDAOProposal contract directly
XFAIL tests/tests/test_trusted_dao.py::test_proposal_replace
  Second event argument should be the address of the replacement member
XFAIL tests/tests/test_trusted_dao.py::test_upgrade_rpl_token_contract_with_empty_abi
  Should not be able to upgrade RPL token contract
XFAIL tests/tests/test_trusted_dao.py::test_upgrade_abi_with_existing_abi
  Should not succeed in upgrading abi when new and old abi are the same
XFAIL tests/tests/test_trusted_dao_reporting.py::test_malicious_withdrawal_reporting[True]
  Reported issue: replaced DAO members can submit the same exits, increasing voting power
XFAIL tests/tests/test_trusted_dao_reporting.py::test_malicious_withdrawal_reporting[False]
  Reported issue: replaced DAO members can submit the same exits, increasing voting power
================= 130 passed, 11 xfailed in 505.09s (0:08:25) ===============
```

Appendix B Round Two Test Suite

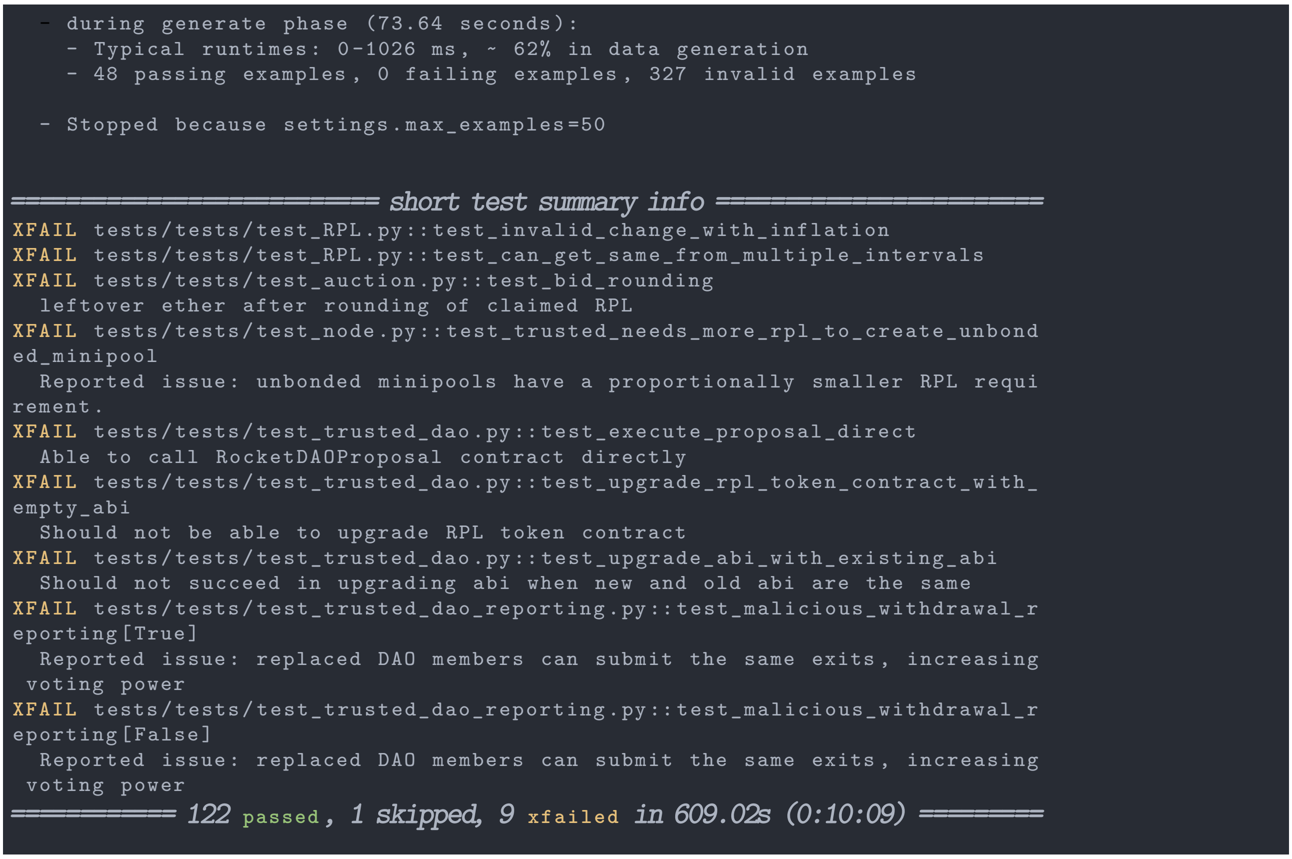
A non-exhaustive list of tests were constructed to aid this security review and are provided alongside this document. The `brownie` framework was used to perform these tests and the output is given below.

This display includes changed test results, reflecting identified issues that were fixed, as well as new tests implemented in the subsequent round of review.

```
Brownie v1.14.3 - Python development framework for Ethereum

======================== test session starts ======================

test_RPL.py:test_rpl_deploy_vars                                             PASSED   [0%]
test_RPL.py:test_daosettings_match_views                                     PASSED   [1%]
test_RPL.py:test_inflation_calculate_changes                                 PASSED   [2%]
test_RPL.py:test_invalid_change_with_inflation                               XFAIL    [3%]
test_RPL.py:test_inflation_no_mint_tokens_before_interval                    PASSED   [3%]
test_RPL.py:test_inflation_can_mint_after_default_interval                   PASSED   [4%]
test_RPL.py:test_inflation_can_mint_after_set_interval                       PASSED   [5%]
test_RPL.py:test_cannot_mint_after_minting_in_same_interval                  PASSED   [6%]
test_RPL.py:test_cannot_swap_tokens_if_not_assigned                          PASSED   [6%]
test_RPL.py:test_cannot_mint_without_allowance                               PASSED   [7%]
test_RPL.py:test_cannot_swap_more_than_allowance                             PASSED   [8%]
test_RPL.py:test_can_swap_tokens_balance                                     PASSED   [9%]
test_RPL.py:test_can_swap_tokens_in_multiple_calls                           PASSED   [9%]
test_RPL.py:test_cannot_swap_greater_than_balance                            PASSED   [10%]
test_RPL.py:test_can_swap_during_after_inflation                             PASSED   [11%]
test_RPL.py:test_can_get_same_from_two_intervals                             PASSED   [12%]
test_RPL.py:test_can_get_same_from_multiple_intervals                        XFAIL    [12%]
test_auction.py:test_auction_settings                                        PASSED   [13%]
test_auction.py:test_create_lot_insufficient_balance                         PASSED   [14%]
test_auction.py:test_create_lot_disabled                                     PASSED   [15%]
test_auction.py:test_create_lot                                              PASSED   [15%]
test_auction.py:test_place_bid_zero_value                                    PASSED   [16%]
test_auction.py:test_place_bid_nonexistent_lot                               PASSED   [17%]
test_auction.py:test_place_bid_bidding_disabled                              PASSED   [18%]
test_auction.py:test_place_bid_lot_expired                                   PASSED   [18%]
test_auction.py:test_place_bid_allocation_exhausted                          PASSED   [19%]
test_auction.py:test_place_bid_refund_excess                                 PASSED   [20%]
test_auction.py:test_place_bid                                               PASSED   [21%]
test_auction.py:test_claim_bid_nonexistent_lot                               PASSED   [21%]
test_auction.py:test_claim_bid_no_claim                                      PASSED   [22%]
test_auction.py:test_claim_bid                                               PASSED   [23%]
test_auction.py:test_bid_rounding                                            XFAIL    [24%]
test_auction.py:test_recover_unclaimed_bidding_not_concluded                 PASSED   [25%]
test_auction.py:test_recover_unclaimed_no_rpl                                PASSED   [25%]
test_auction.py:test_recover_unclaimed                                       PASSED   [26%]
test_auction.py:test_full_auction                                            PASSED   [27%]
test_dao_protocol.py:test_guardian_bootstrap_settings                        PASSED   [28%]
test_dao_protocol.py:test_non_guardian_bootstrap_settings                    PASSED   [28%]
test_dao_protocol.py:test_guardian_disable_bootstrap_mode                    PASSED   [29%]
test_dao_protocol.py:test_guardian_change_setting_bootstrap_disabled         PASSED   [30%]
test_deploy.py:test_registered_instance_registration[rocketVault]            PASSED   [31%]
test_deploy.py:test_registered_instance_registration[rocketAuctionManager]   PASSED   [31%]
test_deploy.py:test_registered_instance_registration[rocketDepositPool]      PASSED   [32%]
test_deploy.py:test_registered_instance_registration[rocketMinipoolDelegate] PASSED   [33%]
test_deploy.py:test_registered_instance_registration[rocketMinipoolFactory]  PASSED   [34%]
test_deploy.py:test_registered_instance_registration[rocketMinipoolManager]  PASSED   [34%]
test_deploy.py:test_registered_instance_registration[rocketMinipoolQueue]    PASSED   [35%]
test_deploy.py:test_registered_instance_registration[rocketMinipoolStatus]   PASSED   [36%]
test_deploy.py:test_registered_instance_registration[rocketNetworkBalances]  PASSED   [37%]
test_deploy.py:test_registered_instance_registration[rocketNetworkFees]      PASSED   [37%]
test_deploy.py:test_registered_instance_registration[rocketNetworkPrices]    PASSED   [38%]
test_deploy.py:test_registered_instance_registration[rocketNetworkWithdrawal PASSED   [39%]
]
```

```
test_deploy.py:test_registered_instance_registration[rocketRewardsPool]        PASSED    [40%]
test_deploy.py:test_registered_instance_registration[rocketClaimDAO]           PASSED    [40%]
test_deploy.py:test_registered_instance_registration[rocketClaimNode]          PASSED    [41%]
test_deploy.py:test_registered_instance_registration[rocketClaimTrustedNode]   PASSED    [42%]
test_deploy.py:test_registered_instance_registration[rocketNodeDeposit]        PASSED    [43%]
test_deploy.py:test_registered_instance_registration[rocketNodeManager]        PASSED    [43%]
test_deploy.py:test_registered_instance_registration[rocketNodeStaking]        PASSED    [44%]
test_deploy.py:test_registered_instance_registration[rocketDAOProposal]        PASSED    [45%]
test_deploy.py:test_registered_instance_registration[rocketDAONodeTrusted]     PASSED    [46%]
test_deploy.py:test_registered_instance_registration[rocketDAONodeTrustedPro   PASSED    [46%]
posals]
test_deploy.py:test_registered_instance_registration[rocketDAONodeTrustedAct   PASSED    [47%]
ions]
test_deploy.py:test_registered_instance_registration[rocketDAONodeTrustedUpg   PASSED    [48%]
rade]
test_deploy.py:test_registered_instance_registration[rocketDAONodeTrustedSet   PASSED    [49%]
tingsMembers]
test_deploy.py:test_registered_instance_registration[rocketDAONodeTrustedSet   PASSED    [50%]
tingsProposals]
test_deploy.py:test_registered_instance_registration[rocketDAOProtocol]        PASSED    [50%]
test_deploy.py:test_registered_instance_registration[rocketDAOProtocolPropos   PASSED    [51%]
als]
test_deploy.py:test_registered_instance_registration[rocketDAOProtocolAction   PASSED    [52%]
s]
test_deploy.py:test_registered_instance_registration[rocketDAOProtocolSettin   PASSED    [53%]
gsInflation]
test_deploy.py:test_registered_instance_registration[rocketDAOProtocolSettin   PASSED    [53%]
gsRewards]
test_deploy.py:test_registered_instance_registration[rocketDAOProtocolSettin   PASSED    [54%]
gsAuction]
test_deploy.py:test_registered_instance_registration[rocketDAOProtocolSettin   PASSED    [55%]
gsNode]
test_deploy.py:test_registered_instance_registration[rocketDAOProtocolSettin   PASSED    [56%]
gsNetwork]
test_deploy.py:test_registered_instance_registration[rocketDAOProtocolSettin   PASSED    [56%]
gsDeposit]
test_deploy.py:test_registered_instance_registration[rocketDAOProtocolSettin   PASSED    [57%]
gsMinipool]
test_deploy.py:test_registered_instance_registration[rocketTokenRPLFixedSupp   PASSED    [58%]
ly]
test_deploy.py:test_registered_instance_registration[rocketTokenRETH]          PASSED    [59%]
test_deploy.py:test_registered_instance_registration[rocketTokenRPL]           PASSED    [59%]
test_deploy.py:test_registered_instance_registration[addressQueueStorage]      PASSED    [60%]
test_deploy.py:test_registered_instance_registration[addressSetStorage]        PASSED    [61%]
test_deploy.py:test_registered_instance_registration[casperDeposit]            PASSED    [62%]
test_deploy.py:test_registered_abi_only                                        PASSED    [62%]
test_deploy.py:test_storage_should_be_initialised                              PASSED    [63%]
test_deploy.py:test_owner_should_not_be_registered_as_a_contract               PASSED    [64%]
test_minipool_reward.py:test_get_minipool_reward_positive[0]                   PASSED    [65%]
test_minipool_reward.py:test_get_minipool_reward_positive[16 ether]            PASSED    [65%]
test_minipool_reward.py:test_get_minipool_reward_positive[32 ether]            PASSED    [66%]
test_minipool_reward.py:test_get_minipool_reward_negative[0]                   PASSED    [67%]
test_minipool_reward.py:test_get_minipool_reward_negative[16 ether]            PASSED    [68%]
test_minipool_reward.py:test_get_minipool_reward_negative[32 ether]            PASSED    [68%]
test_node.py:test_cannot_create_minipool_without_rpl_stake                     PASSED    [69%]
test_node.py:test_can_create_minipool_when_staking_rpl[MinipoolDepositCls.FU   PASSED    [70%]
LL]
test_node.py:test_can_create_minipool_when_staking_rpl[MinipoolDepositCls.HA   PASSED    [71%]
LF]
test_node.py:test_normal_node_cannot_create_unbonded_minipool                  PASSED    [71%]
test_node.py:test_trusted_node_cannot_create_without_staked_rpl[MinipoolDepo   PASSED    [72%]
sitCls.FULL]
test_node.py:test_trusted_can_create_minipool_with_any_deposit_amount[Minipo   PASSED    [73%]
olDepositCls.FULL]
test_node.py:test_trusted_node_cannot_create_without_staked_rpl[MinipoolDepo   PASSED    [74%]
sitCls.HALF]
test_node.py:test_trusted_can_create_minipool_with_any_deposit_amount[Minipo   PASSED    [75%]
olDepositCls.HALF]
test_node.py:test_trusted_node_cannot_create_without_staked_rpl[MinipoolDepo   PASSED    [75%]
sitCls.EMPTY]
```

```
test_node.py:test_trusted_can_create_minipool_with_any_deposit_amount[Minipo  PASSED    [76%]
olDepositCls.EMPTY]
test_node.py:test_trusted_needs_more_rpl_to_create_unbonded_minipool         XFAIL     [77%]
test_node.py:test_node_withdrawal                                             PASSED    [78%]
test_node.py:test_payout_before_refund                                        PASSED    [78%]
test_rETH.py:test_rETH_deployment_params                                      PASSED    [79%]
test_rETH.py:test_rETH_zero_storage                                           PASSED    [80%]
test_rETH.py:test_only_network_can_deposit                                    PASSED    [81%]
test_rETH.py:test_deposit_reward_respects_updates                             PASSED    [81%]
test_rETH.py:test_can_get_correct_eth_value_at_start                          PASSED    [82%]
test_storage.py:test_can_retrieve_some_stored_uint                            PASSED    [83%]
test_storage.py:test_can_retrieve_some_stored_string                          PASSED    [84%]
test_storage.py:test_can_retrieve_some_stored_address                         PASSED    [84%]
test_storage.py:test_can_retrieve_some_stored_bytes                           PASSED    [85%]
test_storage.py:test_can_retrieve_some_stored_bool                            PASSED    [86%]
test_storage.py:test_can_retrieve_some_stored_int                             PASSED    [87%]
test_storage.py:test_can_retrieve_some_stored_bytes32                         PASSED    [87%]
test_storage.py:test_different_types_are_independent                          PASSED    [88%]
test_storage.py:test_locking                                                  PASSED    [89%]
test_trusted_dao.py:test_create_proposal_and_execute                          PASSED    [90%]
test_trusted_dao.py:test_execute_proposal_empty_calldata                      PASSED    [90%]
test_trusted_dao.py:test_cancel_proposal                                      PASSED    [91%]
test_trusted_dao.py:test_execute_proposal_direct                              XFAIL     [92%]
test_trusted_dao.py:test_action_join_required                                 PASSED    [93%]
test_trusted_dao.py:test_proposal_leave                                       PASSED    [93%]
test_trusted_dao.py:test_proposal_replace                                     SKIPPED   [94%]
test_trusted_dao.py:test_proposal_kick                                        PASSED    [95%]
test_trusted_dao.py:test_challenge_node                                       PASSED    [96%]
test_trusted_dao.py:test_upgrade_rpl_token_contract_with_empty_abi            XFAIL     [96%]
test_trusted_dao.py:test_upgrade_abi_with_existing_abi                        XFAIL     [97%]
test_trusted_dao_reporting.py:test_correct_withdrawal                         PASSED    [98%]
test_trusted_dao_reporting.py:test_malicious_withdrawal_reporting[True]       XFAIL     [99%]
test_trusted_dao_reporting.py:test_malicious_withdrawal_reporting[False]      XFAIL     [100%]
============================ Hypothesis Statistics ============================

test_minipool_reward.py:test_get_minipool_reward_positive[0]:

  - during reuse phase (0.02 seconds):
    - Typical runtimes: ~ 22ms, ~ 3% in data generation
    - 1 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (14.35 seconds):
    - Typical runtimes: 0-344 ms, ~ 23% in data generation
    - 49 passing examples, 0 failing examples, 17 invalid examples

  - Stopped because settings.max_examples=50


test_minipool_reward.py:test_get_minipool_reward_positive[16 ether]:

  - during reuse phase (0.27 seconds):
    - Typical runtimes: ~ 266ms, ~ 0% in data generation
    - 1 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (13.96 seconds):
    - Typical runtimes: 0-309 ms, ~ 24% in data generation
    - 49 passing examples, 0 failing examples, 17 invalid examples

  - Stopped because settings.max_examples=50


test_minipool_reward.py:test_get_minipool_reward_positive[32 ether]:

  - during reuse phase (0.39 seconds):
    - Typical runtimes: ~ 358ms, ~ 0% in data generation
    - 1 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (13.99 seconds):
    - Typical runtimes: 0-326 ms, ~ 23% in data generation
    - 49 passing examples, 0 failing examples, 17 invalid examples
  - Stopped because settings.max_examples=50


test_minipool_reward.py:test_get_minipool_reward_negative[0]:

  - during reuse phase (0.34 seconds):
    - Typical runtimes: ~ 334ms, ~ 0% in data generation
    - 1 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (13.83 seconds):
    - Typical runtimes: 0-307 ms, ~ 23% in data generation
    - 49 passing examples, 0 failing examples, 17 invalid examples

  - Stopped because settings.max_examples=50


test_minipool_reward.py:test_get_minipool_reward_negative[16 ether]:

  - during reuse phase (0.32 seconds):
    - Typical runtimes: ~ 295ms, ~ 0% in data generation
    - 1 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (13.81 seconds):
    - Typical runtimes: 0-300 ms, ~ 23% in data generation
    - 49 passing examples, 0 failing examples, 17 invalid examples

  - Stopped because settings.max_examples=50


test_minipool_reward.py:test_get_minipool_reward_negative[32 ether]:

  - during reuse phase (0.30 seconds):
    - Typical runtimes: ~ 276ms, ~ 0% in data generation
    - 1 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (14.47 seconds):
    - Typical runtimes: 0-330 ms, ~ 24% in data generation
    - 49 passing examples, 0 failing examples, 17 invalid examples

  - Stopped because settings.max_examples=50


test_storage.py:test_can_retrieve_some_stored_uint:

  - during reuse phase (0.24 seconds):
    - Typical runtimes: ~ 237ms, ~ 0% in data generation
    - 1 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (3.50 seconds):
    - Typical runtimes: 344-456 ms, ~ 0% in data generation
    - 9 passing examples, 0 failing examples, 0 invalid examples

  - Stopped because settings.max_examples=10


test_storage.py:test_can_retrieve_some_stored_string:

  - during reuse phase (0.57 seconds):
    - Typical runtimes: 108-456 ms, ~ 0% in data generation
    - 2 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (3.08 seconds):
    - Typical runtimes: 0-484 ms, ~ 40% in data generation
    - 8 passing examples, 0 failing examples, 6 invalid examples

  - Stopped because settings.max_examples=10


test_storage.py:test_can_retrieve_some_stored_address:
  - during reuse phase (0.11 seconds):
    - Typical runtimes: ~ 86ms, ~ 0% in data generation
    - 1 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (3.64 seconds):
    - Typical runtimes: 329-505 ms, ~ 0% in data generation
    - 9 passing examples, 0 failing examples, 0 invalid examples

  - Stopped because settings.max_examples=10


test_storage.py:test_can_retrieve_some_stored_bytes:

  - during reuse phase (0.17 seconds):
    - Typical runtimes: ~ 145ms, ~ 0% in data generation
    - 1 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (3.67 seconds):
    - Typical runtimes: 2-475 ms, ~ 10% in data generation
    - 9 passing examples, 0 failing examples, 1 invalid examples

  - Stopped because settings.max_examples=10


test_storage.py:test_can_retrieve_some_stored_bool:

  - during reuse phase (0.13 seconds):
    - Typical runtimes: ~ 127ms, ~ 0% in data generation
    - 1 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (3.62 seconds):
    - Typical runtimes: 355-502 ms, ~ 0% in data generation
    - 9 passing examples, 0 failing examples, 0 invalid examples

  - Stopped because settings.max_examples=10


test_storage.py:test_can_retrieve_some_stored_int:

  - during reuse phase (0.11 seconds):
    - Typical runtimes: ~ 99ms, ~ 1% in data generation
    - 1 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (3.35 seconds):
    - Typical runtimes: 332-389 ms, ~ 0% in data generation
    - 9 passing examples, 0 failing examples, 0 invalid examples

  - Stopped because settings.max_examples=10


test_storage.py:test_can_retrieve_some_stored_bytes32:

  - during reuse phase (0.08 seconds):
    - Typical runtimes: ~ 79ms, ~ 0% in data generation
    - 1 passing examples, 0 failing examples, 0 invalid examples

  - during generate phase (3.27 seconds):
    - Typical runtimes: 345-400 ms, ~ 0% in data generation
    - 9 passing examples, 0 failing examples, 0 invalid examples

  - Stopped because settings.max_examples=10


test_storage.py:test_different_types_are_independent:

  - during reuse phase (1.27 seconds):
    - Typical runtimes: 481-784 ms, ~ 0% in data generation
    - 2 passing examples, 0 failing examples, 0 invalid examples
```



```
  - during generate phase (73.64 seconds):
    - Typical runtimes: 0-1026 ms, ~ 62% in data generation
    - 48 passing examples, 0 failing examples, 327 invalid examples

  - Stopped because settings.max_examples=50


=========================== short test summary info ========================
XFAIL tests/tests/test_RPL.py::test_invalid_change_with_inflation
XFAIL tests/tests/test_RPL.py::test_can_get_same_from_multiple_intervals
XFAIL tests/tests/test_auction.py::test_bid_rounding
  leftover ether after rounding of claimed RPL
XFAIL tests/tests/test_node.py::test_trusted_needs_more_rpl_to_create_unbond
ed_minipool
  Reported issue: unbonded minipools have a proportionally smaller RPL requi
rement.
XFAIL tests/tests/test_trusted_dao.py::test_execute_proposal_direct
  Able to call RocketDAOProposal contract directly
XFAIL tests/tests/test_trusted_dao.py::test_upgrade_rpl_token_contract_with_
empty_abi
  Should not be able to upgrade RPL token contract
XFAIL tests/tests/test_trusted_dao.py::test_upgrade_abi_with_existing_abi
  Should not succeed in upgrading abi when new and old abi are the same
XFAIL tests/tests/test_trusted_dao_reporting.py::test_malicious_withdrawal_r
eporting[True]
  Reported issue: replaced DAO members can submit the same exits, increasing
 voting power
XFAIL tests/tests/test_trusted_dao_reporting.py::test_malicious_withdrawal_r
eporting[False]
  Reported issue: replaced DAO members can submit the same exits, increasing
 voting power
============ 122 passed, 1 skipped, 9 xfailed in 609.02s (0:10:09) =========
```

Appendix C Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
			Likelihood	

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

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: `https://blog.sigmaprime.io/solidity-security.html`. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: `http://www.dasp.co/`. [Accessed 2018].

[3] Docker. Using secrets in docker compose, Available: `https://docs.docker.com/engine/swarm/secrets/#use-secrets-in-compose`.

[4] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 2014. Petersburg Version 41c1837 – Updated February 2021, Available: `https://ethereum.github.io/yellowpaper/paper.pdf`.

[5] Sigma Prime. Solidity Security - Delegatecall. Blog, 2018, Available: `https://blog.sigmaprime.io/solidity-security.html#delegatecall`. [Accessed 2018].