Rocketpool

1 Executive Summary

Date	April 2021
Lead Auditor	Dominik Muhs
Co-auditors	Martin Ortner, David Oz Kashi

This report presents the results of our engagement with **Rocket Pool** to review **the smart contract system**, **the go language bindings** for smart contract interaction, and the **Smart Node** implementation.

The review was conducted over five weeks, from **March 8, 2021** to **April 9, 2021**. A total of 40 person-days were spent.

Due to the audit's extend, covering more than 6000 solidity source lines of code (ca 3.5k normalized) with 47 logic contracts in a complex smart contract system. A Golang implementation of a RocketPool node application consuming the smart contracts, the results represented with this report are to be interpreted as a **best effort** review given the broad scope and time-boxed nature of this engagement.

Technical documentation other than inline source code or blog posts was not available for this review. It is highly recommended that technical documentation and a precise specification for the main components that comprise the system be created. For example, a security documentation that outlines risks and potential threats to the system, technical system, and design documentation that outlines the main components and how they interact, in what places value is stored, and how to safely upgrade/migrate parts of the system. Furthermore, diagrams for high-level interaction flows and outlines that describe the essential workings of the main components (Settings, Vault, Node Management, Minipools and Staking, Rewards, User Staking, Auctions, DAO Member responsibilities, Oracle functionality, DAO proposals, and risks). Ultimately, it is paramount that before the system goes live, the team establishes incident response readiness, having worst-case scenarios and risks assessed, and risk mitigation and incident treatment playbooks prepared.

1.1 Timeline

- During the first week, the assessment team spent time understanding the system, map the attack surface, and explore potential high-risk areas. The assessment team split up the efforts with one part of the team investigating the off-chain components and mapping the smart contract system.
- The second week was spent assessing the smart contract system (vault, storage, tokens, general view on node management) and the interaction with the off-chain elements.
- After a one-week hiatus, the assessment team continued to assess the node- and minipool-management functionality in the smart contract system and the high-level interaction with the various custom tokens in the system.
- The fourth week continued with reviewing tokens/rewards/auctions and transitioned into reviewing the DAO implementation.

Given the limited time available for this review and the amount of findings listed in this report, combined with the sparse availability of documentation it is suggested that further security reviews be conducted.

Update: 27 May 20201 - Mitigations

The report was updated to reflect mitigations implemented for the findings. An additional `5` person days (in the week of May 24 - May 27) were spent to conduct the review, focusing on reviewing the changes that were implemented addressing the specific findings. As with every project that undergoes significant changes it is recommended to conduct a complete re-assessment of the changed system.

2 Scope

Commit Hash	Repository
`44cbf038b97abffa91058cebb2f604220996e641`	https://github.com/rocket-pool/rocketpool/tree/v2.5-Tokenomics-updates
`439f0a2e0db7110fef424361a49df2a0b3cb1a5c`	https://github.com/rocket-pool/rocketpool-go/tree/v2.5-Tokenomics
`7a71915bdb443efbe3d2179d0f6e9cf61f56083e`	https://github.com/rocket-pool/smartnode/tree/v2.5-Tokenomics

2.1 Objectives

Together with the Rocket Pool team, we identified the following priorities for our review:

1. Ensure that the system is implemented consistently with the intended functionality and without unintended edge cases.

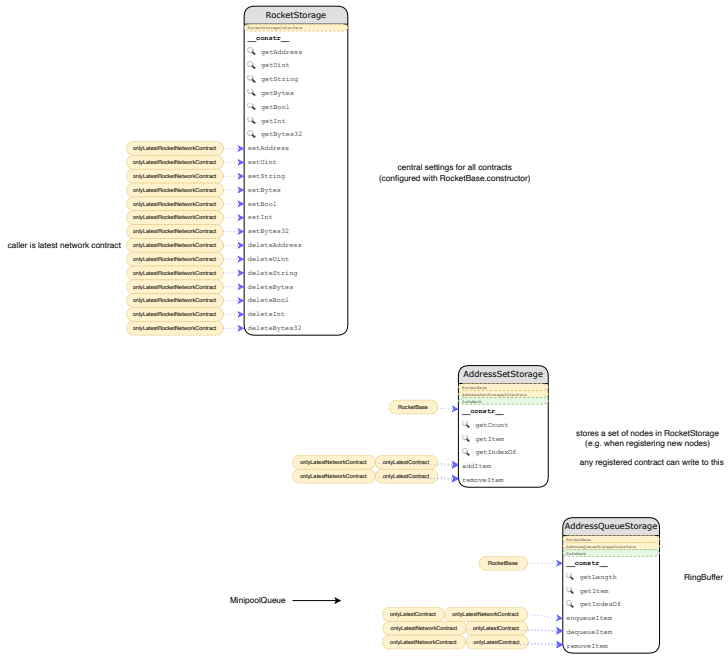
2. Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Best Practices, and the Smart Contract Weakness Classification Registry.
3. Identify security vulnerabilities in the off-chain components

3 System Overview

This section describes the top-level/deployable contracts, their inheritance structure and interfaces, actors, permissions and important contract interactions of the system under review. Please refer to Section 4 - Security Specification for a security-centric view on the system.

Contracts are depicted as boxes. Public reachable interface methods are outlined as rows in the box. The 🔍 icon indicates that a method is declared as non-state-changing (view/pure) while other methods may change state. A yellow dashed row at the top of the contract shows inherited contracts. A green dashed row at the top of the contract indicates that that contract is used in a usingFor declaration. Modifiers used as ACL are connected as yellow bubbles in front of methods.

3.1 Global Storage



The centralized settings and state storage

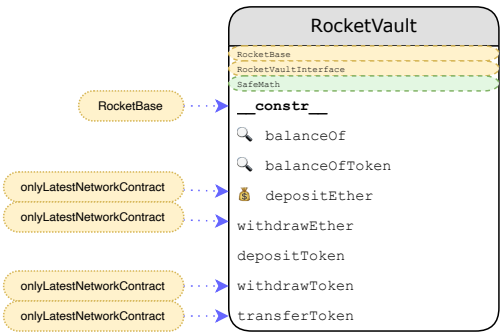
For upgrading purposes the system centralizes almost all settings into a one `RocketStorage` contract. This contract address is then passed to other components of the system on initialization. This generalized contract allows to store various different types of data similar to the `EternalStorage` solidity pattern.

All setter methods are access control protected, however, it is concerning, that, the access restriction is very broad, allowing any registered contract in the system to change storage values even if they do not belong to this specific contract. This might be a risk to the system.

The `AddressSetStorage` is a custom implementation of a `set` type list, which is also stored in the `RocketStorage` and the `AddressQueueStorage` is an implementation for a `Queue` type. While the development team noted that the queue is meant to implement a ring-buffer, it is actually initialized with an unlimited capacitiy `uint256 public capacity = 2 ** 255;` which might be problematic (GasDoS).

After deployment of the contract anyone can theoretically set any value in the contract. This issue is listed as a security issue in the **Findings** section.

3.2 Vault

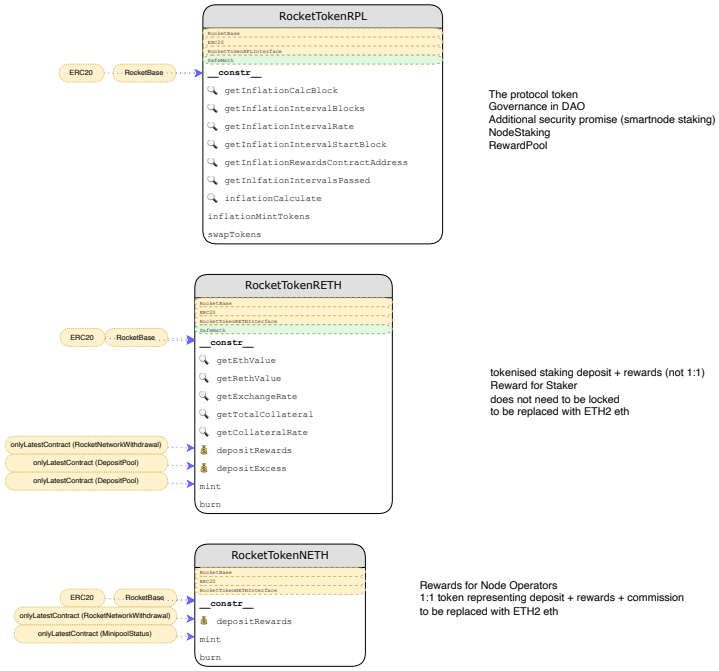


The centralized store of value

The vault serves as a centralized store of value and is likely existent in the system to facilitate an easy way to migrate value when upgrading contracts. The vault may store `ETH` or `ERC20` compliant tokens.

All but one method are access control protected and only allow registered system contracts to interact with them. The `depositToken` method, however, allows anyone to make the vault pull-in any token (any address, no whitelist) and account it to one of the registered system tokens (by name). Tokens or `ETH` can only be withdrawn or transferred by the corresponding contract registered in the system and only for the balance accounted to them.

3.3 Tokens



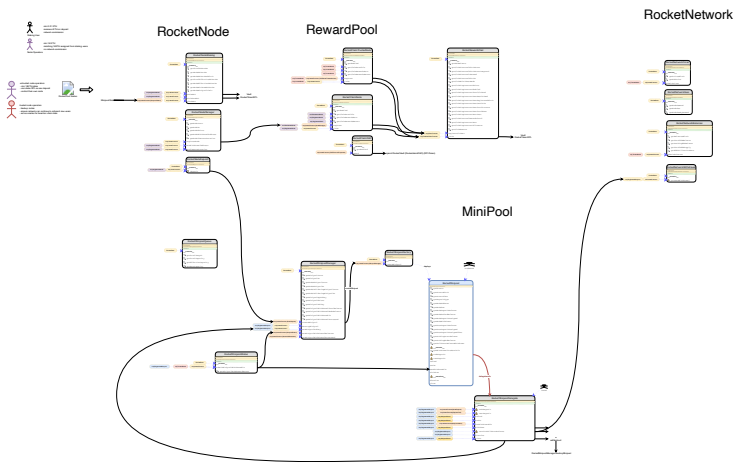
The System's Tokens - RPL, RETH, NETH

The network is using three custom `ERC20` token implementations.

- `RocketTokenNETH`
  - Reward for Node Operators
  - 1:1 Token representing deposit + rewards + commission
  - To be replaced with ETH2 ETH in the future
  - `NETH` for the node reward is minted to the minipool when it is changing state to withdrawable
  - `NETH`, an equivalent of the nodes balance in a minipool, can be withdrawn from the minipool to the nodes withdrawal address when ETH2 staking rewards are received
  - `NETH` can be burned for `ETH` in the token contract
- `RocketTokenRETH`
  - Tokenized staking deposit and rewards
  - Reward for staking nodes
  - To be replaced with ETH2 ETH in the future
  - `ETH` can be deposited for `RETH` in `RocketDepositPool`. This `ETH` will then be pooled into minipools as the user deposit side (16-32ETH). Users receive `RETH` that represents their share of the pooled funds. Rewards are fed back into the `RETH` contract.
  - `RETH` can be burned for `ETH` in the token contract
  - the `RETH` price is set by trusted/oracle nodes (should be a snapshot of the token price; this may allow for manipulations)
  - `RETH` - the token representing the pooled user deposit - is provided to the `RETH` token contract or recycled with the deposit pool when a minipool withdrawal is processed. This should increase the price of `RETH` and passively reward `RETH` holders for providing pooled user deposits to the system.

- `RocketTokenRPL`
  - The RocketPool protocol token
  - Governance token
  - An additional security promise that registered nodes can stake and which may get slashed if they return no staking rewards
  - Allows nodes to stake RPL and earn rewards based on their share of the total amount staked (RewardPool)
  - Inflationary Token - Rewards are minted with a predefined APY of about 5%
  - This token is used in `RocketAuctionManager`, `RocketDaoNodeTrustedActions`, `RocketNodeStaking`, `RocketRewardsPool`, and `RocketClaimDAO`
  - `RPL` can be bought with `ETH` at a discount from `RocketAuctionManager` and more `RPL` may be rewarded to `RPL` stakers via the `RocketRewardsPool`
  - `RPL` can be staked to receive rewards and provide additional security promises via `RocketNodeStaking`
  - `RPL` can be spend by the DAO via `RocketClaimDAO`
  - `RPL` must be provided as a bond for joining the `DAO`

3.4 Node Management, Minipool, Rewards & Auctions



The node flows

Becoming a registered node

Ethereum addresses can register as “registered nodes” via `RocketNodeManager`. There is no fee attached to becoming a registered node. As a side-effect, the node is also registered in the `RocketRewardsPool` to receive rewards if they stake `RPL`. Note that a node cannot unregister from the rewards pool, however, they can reduce their stake to `0` which basically entitles them to no rewards at all.

Ideally, the node would also set a `withdrawalAddress` which is the address rewards (from the reward pool and minipool) are paid out to. A node can also set its own `TimezoneLocation` which basically gives other users a reference about the geographical region the

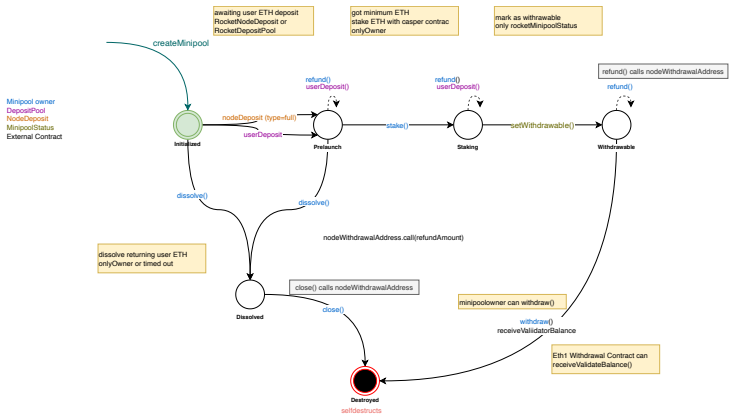
node is from. The timezone can be arbitrary, falsely stated, or even an invalid string.

The node is now registered with the system and can create a minipool.

Creating a minipool

- A registered node can now deposit either the halfDepositAmount (16ETH) or the fullDepositAmount (32ETH) via `RocketNodeDeposit.deposit`. A trusted/oracle (DAO) node may also provide the emptyDespositAmount (0ETH) if the capacity for an empty minipool is available.
- Invoking the `RocketNodeDeposit.deposit` creates a new minipool via the `RocketMinipoolManager` and `RocketMinipoolFactory`. The nodeDeposit provided with the call is forwarded to the newly created minipool via `MinipoolDelegate.nodeDeposit` which records the amount of `ETH` contributed by the node. The caller is set to be the minipool owner and is responsible to perform certain actions progressing the minipool through the various states.
- The deployed minipool contract is bound to that node address and is basically to large extend a proxy that delegatecalls to an implementation called `RocketMinipoolDelegate`.
- The newly created minipool is put into the `RocketMinipoolQueue` where it is due to be funded with user funds.
- The minipool owner can provide an additional security guarantee by staking RPL on the minipool. This amount may get slashed if the pool returns less rewards than the user deposit provided.
- The minipool is now in the `initialized` state.

The following diagram guides through the various states a minipool can be in.



Minipool state diagram and transitions

Initialized

- The minipool was newly created and an initial nodeDeposit was provided.
- If initially the node provided a full deposit (32ETH) the pool is progressed to the `PRELAUNCH` state as it now contains the necessary 32 ETH required for staking in the ETH2 network.

The pool is still accepting a userDeposit in order to allow the registered node to reclaim half of the provided 32 ETH. Ideally, this would allow a registered trusted node to bootstrap a minipool while no user deposits are available for dispatching to reclaim half of the deposit once user funds are available. The general idea for funding minipools is that $^{16}/_{32}$ ETH are provided by the node and the other 16/32ETH are assigned from the pooled user deposits.

- If initially the node provided an empty deposit (0ETH, basically bootstrapping some pools and waiting for user contributions) or the a halfDeposit (16ETH), the pool i waiting for `userFunds` to be provided for the state to be progressed to `PRELAUNCH`.
- `nodeDeposit` or `userDeposit` can only be provided once and the total funds sum up to 32 ETH, the ETH2 deposit amount.
- A pool can only be `dissolved` by the owner in this state.
- A minipool owner may chose to never dissolve the minipool. If usderDeposits come it the pool might still advance to `Prelaunch` where it can be dissolved after a timeout by anyone if the owner does not proceed.

Prelaunch

- The minipool is fully funded either by the node (fullAmount) or by a mix of user/node deposits.
- The owner can `dissolve` the minipool, effectively recycling the user deposit and in a later step returning the node deposit by `closing` the minipool
- If the node provided a full deposit the operator might wait until additional user funds arrive to `refund` 16 of 32 initially provided ETH. The pool will balance at 16 ETH provided by the user and 16 by the node operator.
- If the minipool holds at least the ETH amount required for staking it can be progressed to the `STAKING` state by calling `stake()`. In this step the node operator provides the validatorPubkey, Signature and deposit root for the ETH2 staking contract. The `32ETH` are then staked with the ETH2 staking contract.

Staking

- The pool still accepts `userDeposit` in case no userDeposit was yet assigned (node provided the full amount required for staking). The node can subsequently `refund()` half of the stake to balance the contribution to 16 ETH by node, 16 ETH by user.
- When the validator node exits from the ETH2 chain/staking, the oracle/trusted nodes observe this event and call `setWithdrawable` on the minipool via a majority vote on `RocketMinipoolStatus`. Note that the contract's balance on the ETH2 chain is set by oracle nodes and a 51% consensus must be reached. Note: Having control of 51% of the DAO nodes allows to set arbitrary balances (or upgrade any parts of the system). The minipool progresses to the `WITHDRAWABLE` state.

Withdrawable

- The minipool operator can refund half of their initial deposit if they provided the full amount for staking and no user funds were contributed to the pool.
- Someone may initiate the payout of funds from the `SystemWithdrawalContractAddress` which are received via `minipool.receive()` and `receiveValidatorBalance` which splits the payment into a user and node share issuing `RocketNETH` to the pool that can be claimed by the node and recycling the user amount into `RocketNETH`.
- The node operator can withdraw the `NETH` balances after a `withdrawDelay` by calling `withdraw()`. withdraw implicitly calls `refund()` in case the operator has not called it yet but was eligible for a refund.
- Note that if `NETH` is withdrawn (subject to a withdrawal delay) before `receiveValidatorBalance` received the funds, no `NETH` may be paid out at all and the `NETH` may be ending up stock at the later destroyed minipool address.
- If `withdraw` and `receiveValidatorBalance` was called, the minipool is destroyed and excess ETH is sent to the vault. Note: this ETH will end up being locked in the vault as it is not accounted for.

Dissolved

- Returns the node balance to the nodeWithdrawalAddress and destroys the minipool sending excess ETH to the vault.

Contributing user funds

- Any ethereum address can contribute `ETH` to the `DepositPool.deposit`. `RETH` is minted to the depositor in return for `ETH`. This `RETH` represents a share of the user funds in circulation. Eventually an assignment of user funds to minipools in the queue is triggered (`minipool.userDeposit`).
- The user can withdraw the `RETH` at any point in time as long as enough collateral is avilable to the `DepositPool` contract (funds stored in the `RETH` contract and unassigned funds from `DepositPool` in vault)
- The price of `RETH` is submitted by trusted/oracle nodes majority vote. (might be stale, can be front-run)

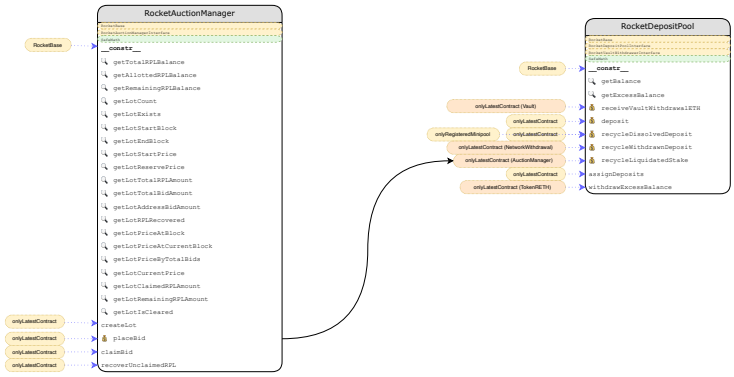
Participating in the rewards distribution

- Any minipool operating node address can participate in the rewards distribution by registering as a node to the `RocketNodeManager` and staking a minimum amount of `RPL` with the `RocketNodeStaking` contract.
- If a minipool's balance is less than the userDeposit, the node operators staked `RPL` is slashed (`max(balance - userDeposit, 1.6ETH)`).
- Rewards are paid out every 14 days. Note: A node cannot claim rewards before waiting at least one period. The stake

only needs to be present just before the reward payout which would allow for manipulation vectors.

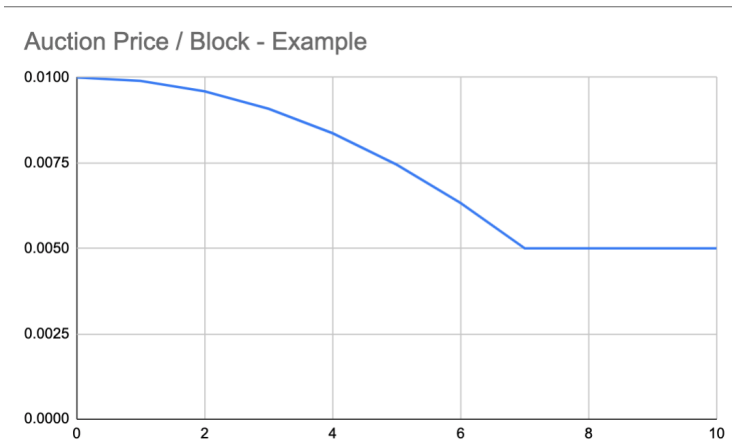
- Registered nodes and Trusted nodes are served from different reward pools that are filled by inflating the `RPL` token for the reward.

Auction



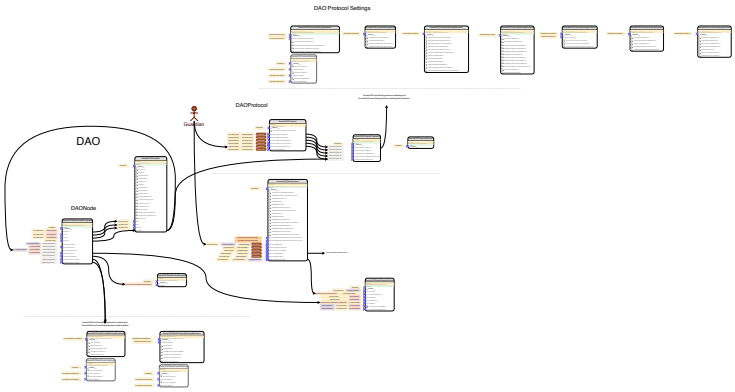
Auction

- `RPL` stake that is slashed from node operators because they returned less than the userDeposit in balance is eventually put to auction with the `RocketAuctionManager` . The auction starts by creating a lot of RPL tokens that starts at 100% the RPL token price and decrements quadratic limiting to 50% the RPL token price with every block. The auction price either establishes at the endPrice (at max 50% the initial RPL price) or at the bidPrice that buys the complete lot if that's reached before the auction ends.



Auction Price Chart - simplified for 7 blocks auction duration, startPrice=0.01 ETH

3.5 Governance / DAO



The RocketDAO and trusted/oracle node flows

The DAO is bootstrapped by a Guardian role that can add any members, upgrade components, and set settings for the DAO. At the end of the bootstrapping phase the Guardian revokes its own permissions by calling `bootstrapDisable` on the `RocketDAOProtocol` and `RocketDAONodeTrusted` contracts. Once the system is in production trusted nodes should verify that the Guardian role revoked its permissions (`getBootstrapModeDisabled()`).

If the DAO membercount falls below a configured threshold (3), any registered node can invite themselves into the DAO. This can be especially risky during the bootstrapping phase or when members are kicked from the DAO as malicious nodes might get a chance to take over the DAO.

Trusted/Oracle nodes perform a variety of activities in the system. They act as oracles for ETH2 events and statis (withdrawals, voluntary leaves, balances), as price oracles for the system token, and they maintain the systems code and configuration. Being able to control 51% of the oracle nodes will be disastrous for the system as it allows this malicious majority to effectively take over the system and pay out any prices/tokens. It is important to understand that a lot of the security of the system shoulders on the trusted/oracle nodes being honest and trustful.

Trusted nodes can propose a variety of changes. They can propose new members to be added, they can propose to leave or get replaced by another trusted node, they can propose to kick other members, propose changes to the systems code (upgrades) or configuration. Members can also challenge each other to proof that they are online. If the challenge fails, the member gets evicted from the DAO. Members can basically challenge each other for free, while registered nodes have to pay to challenge. This also means that members can bully each other by forcing them to spend gas on resolving a challenge (griefing; once a day and if the node fails to respond within 7 days it is evicted). This might give some attack surface for a DAO takeover, especially if the member count drops below the threshold where nodes can invite themselves into the DAO without requiring approval from existing members. It is important to point out that input validation for a lot of settings that can be changed by DAO members is not existent. DAO members may vote on a proposal that may render the system unstable or unusable.

Proposals can be in one of the following states:

- Pending - The vote on the proposal has not started yet (including the startblock)
- Active - The vote on the proposal has started (open)
- Cancelled - The proposal was canceled by the proposer
- Defeated - The proposal did not get enough support (fail, default fall thru case)
- Succeeded - The proposal got enough support (pass) but was not yet executed and can still expire
- Expired - The proposal expired and was not executed but may have been defeated or succeeded or active/pending
- Executed - The proposal was executed

The lifetime of a proposal is defined by three milestones that can be configured when adding a proposal is added, the proposal `startBlock`, the `durationBlocks` and `expiresBlocks`.

A proposal is

- in the `PENDING` state up until including the `_startBlock`: meaning voting is not yet open
- in the `ACTIVE` state up until including the `endBlock`: meaning voting is open
- in the `EXPIRED` state after including the `expiresBlock`: meaning any action on the proposal is closed

4 Security Specification

This section describes the expected behavior of the system under review **from a security perspective**. It is not a substitute for documentation. The purpose of this section is to identify specific security properties and outline trust assumptions. While the security impact of the trust model notes is limited, they contain information that should be taken into account for the system's continued security.

4.1 Actors

The relevant actors are listed below with their respective abilities:

- An Ethereum address that does not hold a role in the system
- Contract Deployer
- Guardian (DAO)
- Regular Node Operators
- Trusted/Oracle Nodes Operators (DAO members)
- Users who have deposited funds into the system

4.2 Trust Assumptions

DAO Members / Trusted Nodes

DAO members can vote to kick others. If, after a kick, there are no more members left to vote on certain actions, votes may get stuck

until a new DAO member is onboarded.

Trusted node operators perform oracle price feed updates. If a price update succeeds once for an extremely high block number, the price feed is permanently stuck as price updates for lower block numbers are not allowed. No check against reasonable bounds is done.

DAO proposals have broad upgrade abilities, including upgrading the contract itself. With enough value locked in the system, this can incentivize trusted DAO members to extract all funds from the system by voting for a malicious upgrade.

Trusted nodes can challenge other nodes free of charge. With the default settings, a challenge can be sent out once per day, with a response window of 7 days. Trusted nodes can challenge other DAO members repeatedly. Consequently, this mechanism can be used for griefing and harassment until the offending member is hopefully removed by other DAO members.

Settings Access Control

Smart contracts registered in the system can change related as well as unrelated settings. This includes proposals accepted by the Trusted Node DAO. The development team should consider separating the settings into trusted regions and restrict each component to only the settings keys it requires to function.

5 Recommendations

5.1 RocketMinipool - getWithdrawalCredentials can be simplified

Description

Consider simplifying the logic in `getWithdrawalCredentials` to `abi.encodePacked(byte(0x01), bytes11(0x0), address(this))`.

Examples

rocketpool-2.5-Tokenomics-updates/contracts/contract/minipool/RocketMinipool.sol:L91-L105

```
function getWithdrawalCredentials() override external view return
    // Parameters
    uint256 credentialsLength = 32;
    uint256 addressLength = 20;
    uint256 addressOffset = credentialsLength - addressLength;
    byte withdrawalPrefix = 0x01;
    // Calculate & return
    bytes memory ret = new bytes(credentialsLength);
    bytes20 addr = bytes20(address(this));
    ret[0] = withdrawalPrefix;
    for (uint256 i = 0; i < addressLength; i++) {
        ret[i + addressOffset] = addr[i];
    }
    return ret;
}
```

5.2 Where possible, a specific contract type should be used rather than `address`

Description

Consider using the best type available in the function arguments and even declaration instead of accepting `address` and later casting it to the correct type.

Examples

This is only one of many examples. The method accepts `address` type arguments while this could already be declared as `RocketStorageInterface _rocketStorageAddress`.

rocketpool-2.5-Tokenomics-updates/contracts/contract/minipool/RocketMinipool.sol:L71-L77

```
constructor(address _rocketStorageAddress, address _nodeAddress,
    // Check parameters
    require(_rocketStorageAddress != address(0x0), "Invalid stora
    require(_nodeAddress != address(0x0), "Invalid node address")
    require(_depositType != MinipoolDeposit.None, "Invalid deposi
    // Initialise RocketStorage
    rocketStorage = RocketStorageInterface(_rocketStorageAddress)
```

`_tokenAddress` can be declared as `IERC20` in arguments and events.

rocketpool-2.5-Tokenomics-updates/contracts/contract/RocketVault.sol:L82-L82

```
IERC20 tokenContract = IERC20(_tokenAddress);
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/RocketVault.sol:L26-L27

```
event TokenDeposited(bytes32 indexed by, address indexed tokenAdd
event TokenWithdrawn(bytes32 indexed by, address indexed tokenAdd
```

`RocketStorageInterface` can be declared in the argument list.

rocketpool-2.5-Tokenomics-updates/contracts/contract/RocketBase.sol:L76-L80

```
/// @dev Set the main Rocket Storage address
constructor(address _rocketStorageAddress) {
    // Update the contract address
    rocketStorage = RocketStorageInterface(_rocketStorageAddress)
}
```

`RocketMinipool` cast to address while it is used as `RocketMinipool` contract by the caller creating the contract.

rocketpool-2.5-Tokenomics-updates/contracts/contract/minipool/RocketMinipoolFactory.sol:L20-L25

```
function createMinipool(address _nodeAddress, MinipoolDeposit _de
    // Create RocketMinipool contract
    address contractAddress = address(new RocketMinipool(address(
    // Return
    return contractAddress;
}
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/node/RocketNodeDeposit.sol:L65-L66

```
address minipoolAddress = rocketMinipoolManager.createMinipool(ms
RocketMinipoolInterface minipool = RocketMinipoolInterface(minipo
```

Pass the `minipool` contract type instead of `address` to the subcall:

rocketpool-2.5-Tokenomics-updates/contracts/contract/minipool/RocketMinipoolStatus.sol:L64-L64

```
setMinipoolWithdrawable(_minipoolAddress, _stakingStartBalance, _
```

5.3 RocketVault - Follow checks-effects-interactions for token withdrawal

Description

`withdrawToken` breaks checks-effects-interactions by calling `token.transfer` before updating the internal accounting.

This may only be problematic when calling out callback tokens (e.g., ERC-777) tokens or when withdrawing from an unsafe `_tokenAddress` (which should never happen).

It is recommended to update the internal account first (analog to `withdrawEth`) and then call out to the potentially untrusted token or token callback receiver.

Examples

rocketpool-2.5-Tokenomics-updates/contracts/contract/RocketVault.sol:L101-L116

```
function withdrawToken(address _withdrawalAddress, address _token
    // Get contract key
    bytes32 contractKey = keccak256(abi.encodePacked(getContractN
    // Get the token ERC20 instance
    IERC20 tokenContract = IERC20(_tokenAddress);
    // Verify this contract has that amount of tokens at a minimum
    require(tokenContract.balanceOf(address(this)) >= _amount, "I
    // Withdraw to the desired address
    require(tokenContract.transfer(_withdrawalAddress, _amount),
    // Update balances
    tokenBalances[contractKey] = tokenBalances[contractKey].sub(_
    // Emit token withdrawn event
    emit TokenWithdrawn(contractKey, _tokenAddress, _amount, bloc
    // Done
    return true;
}
```

`RocketNodeStaking`:

rocketpool-2.5-Tokenomics-updates/contracts/contract/node/RocketNodeStaking.sol:L172-L175

```
rocketVault.withdrawToken(msg.sender, getContractAddress("rocketT
// Update RPL stake amounts
decreaseTotalRPLStake(_amount);
decreaseNodeRPLStake(msg.sender, _amount);
```

5.4 RocketVault - Consider using SafeERC for third party ERC20 interactions or whitelist allowed tokens

Description

SafeERC20 provides a wrapper around ERC20 standard function calls that handles common deviations from ERC20, like missing return values. For example, the vault implementation relies on the `transfer*` functions to return `true` or fail. Well-known but broken ERC20 tokens (USDT, BNB, OMG, ...) might therefore not be safely used with the system.

Anyone can deposit any token right now. This shouldn't be a problem as long as the system isn't using them. If the vault is not meant to be used with unauthorized tokens, it should be considered to implement a token whitelist.

It should be noted that forcefully sent ETH or tokens cannot be reclaimed via the contract.

Examples

rocketpool-2.5-Tokenomics-updates/contracts/contract/RocketVault.sol:L90-L91

```
require(tokenContract.transferFrom(msg.sender, address(this), _am
// Update contract balance
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/RocketVault.sol:L108-L110

```
// Withdraw to the desired address
require(tokenContract.transfer(_withdrawalAddress, _amount), "Roc
// Update balances
```

5.5 Potential Gas Optimizations

Description

This is a batch issue tracking some opportunities to reduce the gas footprint of the contract system. In general, the gas consumption is very high especially because almost all setting read/writes require an external call. For example, this can be problematic if the ethereum network is congested and gas prices rise unpredictably. Registered and Trusted nodes may not be able or willing to perform their duties. Nodes may race to not be the one that executes actions (proposals, setting oracle values) leading to stale prices/balances or minipools that cannot be progressed. Important contract upgrades may not go through because gas consumption in the DAO and upgrading mechanisms may be too high and this is only to express our concerns about the gas footprint.

The following section lists some opportunities for improvement. However, redundant code and excessive gas consumption was seen throughout the codebase.

Examples

RocketVault

- unnecessary balance/allowance checks before `transfer`. The transfer cannot succeed if balance or allowance is insufficient.

rocketpool-2.5-Tokenomics-updates/contracts/contract/RocketVault.sol:L83-L91

```
// Check they can cover the amount
require(tokenContract.balanceOf(msg.sender) >= _amount, "Not enou
// Check they have allowed this contract to send their tokens
require(tokenContract.allowance(msg.sender, address(this)) >= _am
// Get contract key
bytes32 contractKey = keccak256(abi.encodePacked(_networkContract
// Send the tokens to this contract now
require(tokenContract.transferFrom(msg.sender, address(this), _am
// Update contract balance
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/RocketVault.sol:L107-L110

```
require(tokenContract.balanceOf(address(this)) >= _amount, "Insuf
// Withdraw to the desired address
require(tokenContract.transfer(_withdrawalAddress, _amount), "Roc
// Update balances
```

RocketNodeManager

- duplicate “exists” check in manager and indirectly when adding to `addressSetStorage`

rocketpool-2.5-Tokenomics-updates/contracts/contract/node/RocketNodeManager.sol:L63-L63

```
require(!getBool(keccak256(abi.encodePacked("node.exists", msg.se
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/util/AddressSetStorage.sol:L39-L40

```
function addItem(bytes32 _key, address _value) override external
    require(getUint(keccak256(abi.encodePacked(_key, ".index", _v
```

RocketDAOProposal

- `vote` only needs to check for `ProposalState.Active` as this excludes `ProposalState.Succeeded`

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/RocketDAOProposal.sol:L203-L205

```
require(getState(_proposalID) != ProposalState.Succeeded, "Propos
// Check the proposal is in a state that can be voted on
require(getState(_proposalID) == ProposalState.Active, "Voting is
```

RocketDAONodeTrustedUpgrade

- unnecessary check for `_contractAddress != 0` because `oldContractAddress !=0 &&` _contractAddress != oldContractAddress `implies` _contractAddress !=0`

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/node/RocketDAONodeTrustedUpgrade.sol:L51-L54

```
require(oldContractAddress != address(0x0), "Contract does not ex
// Check new contract address
require(_contractAddress != address(0x0), "Invalid contract addre
require(_contractAddress != oldContractAddress, "The contract add
```

- consider checking for `_name.length` instead of comparing with the hash of an empty string

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/node/RocketDAONodeTrustedUpgrade.sol:L69-L71

```
// Check contract name
bytes32 nameHash = keccak256(abi.encodePacked(_name));
require(nameHash != keccak256(abi.encodePacked("")), "Invalid con
```

- `type` should be an enum instead of `string`

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/node/RocketDAONodeTrustedUpgrade.sol:L27-L36

```
function upgrade(string memory _type, string memory _name, string
    // What action are we performing?
    bytes32 typeHash = keccak256(abi.encodePacked(_type));
    // Lets do it!
    if(typeHash == keccak256(abi.encodePacked("upgradeContract"))
    if(typeHash == keccak256(abi.encodePacked("addContract"))) _a
    if(typeHash == keccak256(abi.encodePacked("upgradeABI"))) _up
    if(typeHash == keccak256(abi.encodePacked("addABI"))) _addABI
}
```

- use local var `challengeBlock` instead of wasting gas on a contract call to an external contract

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/node/RocketDAONodeTrustedActions.sol:L239-L239

```
uint256 challengeBlock = getUint(keccak256(abi.encodePacked(daoNa
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/node/RocketDAONodeTrustedActions.sol:L251-L251

```
if(getUint(keccak256(abi.encodePacked(daoNameSpace, "member.chall
```

- order of checks is wasting gas. Checking `depositType` first may bypass the branch earlier and avoid reaching out to `getMemberIsValid` extcall in most cases.

rocketpool-2.5-Tokenomics-updates/contracts/contract/node/RocketNodeDeposit.sol:L54-L61

```
if (rocketDaoNodeTrusted.getMemberIsValid(msg.sender)) {
    // If creating an unbonded minipool, check current unbonded mi
    if (depositType == MinipoolDeposit.Empty) {
        require(rocketDaoNodeTrusted.getMemberUnbondedValidatorCo
    }
}
// Node is not trusted - it cannot create unbonded minipools
else { require(depositType != MinipoolDeposit.Empty, "Only member
```

- duplicate checks `minipoolDeposit.None` is checked in `deposit` and `RocketMinipool.constructor`. also, putting the `Invalid node deposit amount` check into the else branch would save some gas.

rocketpool-2.5-Tokenomics-updates/contracts/contract/node/RocketNodeDeposit.sol:L47-L52

```
MinipoolDeposit depositType = MinipoolDeposit.None;
if (msg.value == rocketDAOProtocolSettingsMinipool.getFullDeposit
else if (msg.value == rocketDAOProtocolSettingsMinipool.getHalfDe
else if (msg.value == rocketDAOProtocolSettingsMinipool.getEmptyD
// Check deposit type is valid
require(depositType != MinipoolDeposit.None, "Invalid node deposi
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/minipool/RocketMinipool.sol:L75-L75

```
require(_depositType != MinipoolDeposit.None, "Invalid deposit ty
```

- `RocketTokenRPL` - duplicate external calls; unnecessary balance/allowance checks because `transferFrom` should bail by definition if the transfer fails. `amount >0 && balance >0` checks could otherwise be combined as well.

rocketpool-2.5-Tokenomics-updates/contracts/contract/token/RocketTokenRPL.sol:L189-L199

```
function swapTokens(uint256 _amount) override external {
    // Valid amount?
    require(_amount > 0, "Please enter valid amount of RPL to swa
    // Check they have a valid amount to swap from
    require(rplFixedSupplyContract.balanceOf(address(msg.sender))
    // Check they can cover the amount
    require(rplFixedSupplyContract.balanceOf(address(msg.sender))
    // Check they have allowed this contract to send their tokens
    uint256 allowance = rplFixedSupplyContract.allowance(msg.send
    // Enough to cover it?
    require(allowance >= _amount, "Not enough allowance given for
```

- `RocketTokenRPL` - unnecessary initialization before assigning a new value

rocketpool-2.5-Tokenomics-updates/contracts/contract/token/RocketTokenRPL.sol:L172-L174

```
uint256 vaultAllowance = 0;
// Get the current allowance for Rocket Vault
vaultAllowance = rplFixedSupplyContract.allowance(rocketVaultAddr
```

- `RocketTokenRPL` - secret of method invocation wastes gas: `inflationRate` could be calculated after `intervalsSinceLastMint` was checked.

rocketpool-2.5-Tokenomics-updates/contracts/contract/token/RocketTokenRPL.sol:L126-L134

```
function inflationCalculate() override public view returns (uint2
    // The inflation amount
    uint256 inflationTokenAmount = 0;
    // Optimisation
    uint256 inflationRate = getInflationIntervalRate();
    // Compute the number of inflation intervals elapsed since the
    uint256 intervalsSinceLastMint = getInlfationIntervalsPassed(
    // Only update  if last interval has passed and inflation rate
    if(intervalsSinceLastMint > 0 && inflationRate > 0) {
```

5.6 Statevariables that cannot change should be declared `constant` or 'immutable'

Description

For example, in `RocketDAONodeTrusted` the statevars `calcBase`, `daoNameSpace`, and `daoMemberMinCount` cannot be updated and should therefore be declared `constant`.

Besides making it obvious that the parameters are immutable, this will save some gas as access to const literals are resolved at compile time and do not require `SLOAD`'s.

Examples

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/node/RocketDAONodeTrusted.sol:L25-L32

```
uint256 private calcBase = 1 ether;

// The namespace for any data stored in the trusted node DAO (do nc
string private daoNameSpace = 'dao.trustednodes';

// Min amount of trusted node members required in the DAO
uint256 private daoMemberMinCount = 3;
```

e.g. should be `uint256 private constant calcBase = 1 ether`.

rocketpool-2.5-Tokenomics-updates/contracts/contract/util/AddressQueueStorage.sol:L20-L20

```
uint256 public capacity = 2 ** 255; // max uint256 / 2
```

`totalInitialSupply` never changes.

rocketpool-2.5-Tokenomics-updates/contracts/contract/token/RocketTokenRPL.sol:L24-L24

```
uint256 public totalInitialSupply = 18000000000000000000000000;
```

5.7 Casting to the contracts own contract type and address can be avoided by calling `this.<function>`

Recommendation

Instead of casting to `IERC20(self)` to force an external call to a contract function from its own address (`msg.sender`) this could be changed to call `this.transfer()` as the `this` keyword forces an external call. Ideally, with a short explanation of why the call needs to come from the contract address (this is already the case with the current code revision)

rocketpool-2.5-Tokenomics-updates/contracts/contract/token/RocketTokenRPL.sol:L202-L205

```
// Initialise itself and send from it's own balance (cant just do a
IERC20 rplInflationContract = IERC20(address(this));
// Transfer from the contracts RPL balance to the user
require(rplInflationContract.transfer(msg.sender, _amount), "Toke
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/token/RocketTokenRPL.sol:L169-L176

```
// Initialise itself and allow from it's own balance (cant just do
IERC20 rplInflationContract = IERC20(address(this));
// This is to prevent an allowance reentry style attack
uint256 vaultAllowance = 0;
// Get the current allowance for Rocket Vault
vaultAllowance = rplFixedSupplyContract.allowance(rocketVaultAddr
// Now allow Rocket Vault to move those tokens, we also need to acc
require(rplInflationContract.approve(rocketVaultAddress, vaultAll
```

5.8 Avoid shadowing inherited names

Recommendation

`allowance` shadows inherited name `IERC20.allowance()`. Consider renaming the local var `allowance` to not overlap with any inherited name.

rocketpool-2.5-Tokenomics-updates/contracts/contract/token/RocketTokenRPL.sol:L197-L199

```
uint256 allowance = rplFixedSupplyContract.allowance(msg.sender,
// Enough to cover it?
require(allowance >= _amount, "Not enough allowance given for tra
```

5.9 Consistent documentation using NatSpec

Description

For consistency, and user- and machine-readability, it is recommended to use the NatSpec format: https://docs.soliditylang.org/en/v0.7.6/natspec-format.html

5.10 Avoid bypassing the solidity type system with unnecessary low-level calls

Description

Low-level calls that are unnecessary for the system should be avoided whenever possible because low-level calls behave differently from a contract-type call. For example, `address.call(abi.encodeWithSelector("fancy(bytes32)", mybytes)` does not verify that a target is actually a contract, while `ContractInterface(address).fancy(mybytes)` does. Additionally, when calling out to functions declared view/pure, the solidity compiler would actually perform a `staticcall` providing additional security guarantees while a low-level call does not. Similarly, return values have to be decoded manually when performing low-level calls.

Note: if a low-level call needs to be performed, consider relying on `Contract.function.selector` instead of encoding using a hardcoded ABI string.

Examples

- cast the returned address to the correct interface class and perform the call

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/node/RocketDAONodeTrusted.sol:L161-L193

```
// Bootstrap mode - If there are less than the required min amount
function bootstrapMember(string memory _id, string memory _email,
    // Ok good to go, lets add them
    (bool success, bytes memory response) = getContractAddress('r
    // Was there an error?
    require(success, getRevertMsg(response));
}


// Bootstrap mode - Uint Setting
function bootstrapSettingUint(string memory _settingContractName,
    // Ok good to go, lets update the settings
    (bool success, bytes memory response) = getContractAddress('r
    // Was there an error?
    require(success, getRevertMsg(response));
}

// Bootstrap mode - Bool Setting
function bootstrapSettingBool(string memory _settingContractName,
    // Ok good to go, lets update the settings
    (bool success, bytes memory response) = getContractAddress('r
    // Was there an error?
    require(success, getRevertMsg(response));
}


// Bootstrap mode - Upgrade contracts or their ABI
function bootstrapUpgrade(string memory _type, string memory _nam
    // Ok good to go, lets update the settings
    (bool success, bytes memory response) = getContractAddress('r
    // Was there an error?
    require(success, getRevertMsg(response));
}
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/protocol/RocketDAOProtocol.sol:L39-L77

```
// Bootstrap mode - Uint Setting
function bootstrapSettingUint(string memory _settingContractName,
    // Ok good to go, lets update the settings
    (bool success, bytes memory response) = getContractAddress('r
    // Was there an error?
    require(success, getRevertMsg(response));
}

// Bootstrap mode - Bool Setting
function bootstrapSettingBool(string memory _settingContractName,
    // Ok good to go, lets update the settings
    (bool success, bytes memory response) = getContractAddress('r
    // Was there an error?
    require(success, getRevertMsg(response));
}

// Bootstrap mode - Address Setting
function bootstrapSettingAddress(string memory _settingContractNa
    // Ok good to go, lets update the settings
    (bool success, bytes memory response) = getContractAddress('r
    // Was there an error?
    require(success, getRevertMsg(response));
}

// Bootstrap mode - Set a claiming contract to receive a % of RPL i
function bootstrapSettingClaimer(string memory _contractName, uin
    // Ok good to go, lets update the rewards claiming contract amo
    (bool success, bytes memory response) = getContractAddress('r
    // Was there an error?
    require(success, getRevertMsg(response));
}

// Bootstrap mode -Spend DAO treasury
function bootstrapSpendTreasury(string memory _invoiceID, address
    // Ok good to go, lets update the rewards claiming contract amo
    (bool success, bytes memory response) = getContractAddress('r
    // Was there an error?
    require(success, getRevertMsg(response));
}
```

Recommendation

When calling out to a contract known in the system, always prefer typed contract calls (interfaces/contract) instead of low-level calls. This is to avoid errors, potentially unchecked return values, have security guarantees provided by the compiler.

5.11 Potential for collisions when writing/reading settings due to keccak(abi.encodePacked())

Description

The system heavily relies on a centralized hash indexed storage. The storage keys are formed with distinct namespace prefixes potentially concatenated with user tainted input. Here's one example of this:

rocketpool-2.5-Tokenomics-updates/contracts/contract/auction/RocketAuctionManager.sol: L187-L192

```
setBool(keccak256(abi.encodePacked("auction.lot.exists", lotIndex
setUint(keccak256(abi.encodePacked("auction.lot.block.start", lot
setUint(keccak256(abi.encodePacked("auction.lot.block.end", lotIn
setUint(keccak256(abi.encodePacked("auction.lot.price.start", lot
setUint(keccak256(abi.encodePacked("auction.lot.price.reserve", l
setUint(keccak256(abi.encodePacked("auction.lot.rpl.total", lotIn
```

`abi.encodePacked` encodes dynamic types in-place without a length prefix and static types will not be padded if they are shorter than 32 bytes. If namespace prefixes are not chosen with care a user might provide a value that overlaps into another settings namespace (after the prefix). Special care should be taken if dynamic or short types are used with `encodePacked` as this might make it easier to force such situations.

Throughout the review, the assessment team has not found any signs of this issue. However, it should be noted that developers must be made aware of this potential problem and it is highly recommended to support the secure development process by tooling that checks for the potential of overlaps in the CI pipeline.

5.12 Use a logging framework to securely print log messages and encode output to `fmt.print*`

Description

Endusers rely on the information displayed by the application and they are basing their action upon it. A successful attempt in manipulating what is displayed to the user by an attacker can be security-critical. It is therefore recommended to use a safe logging framework for log messages and potentially user tainted information that is displayed in the console or used with another presentation layer (web/xss, db/injection, ..).

For example, the golang `log` package nor the `fmt.print*` family encode `CRLF` (and other control chars) before printing them on the console as they are meant to be used with trusted inputs. User tainted information must, therefore, be properly encoded before being passed to these routines.

5.13 Handle recoverable exceptions in off-chain node utilities

Description

A certain amount of interaction is expected from a registered or trusted node operator. For example, the trusted node operator is supposed to provide oracle services, vote on proposals can be challenged by other nodes to prove liveness, or else they'll be evicted from the DAO.

For example, if someone manages to force a golang exception in the trusted node application causing it to terminate, this node may not be able to defeat challenges or participate in the oracle services it is supposed to provide.

Similar to the ETH2 validator application or an ETH1 blockchain node liveness is important and failure to prove liveness may be severely punished or put the network at risk. The worst-case might probably be someone adding information to a smart contract that causes a golang panic in the client implementation. This might put the network in a stale state where oracle provided information becomes stale/outdated providing opportunities to malicious actors to profit from this.

It is, therefore, highly recommend to segment the application into recoverable and non-recoverable zones. If a recoverable exception is detected (e.g. parseInt trying to parse user tainted data and failing to do so, but the application would be able to continue by just skipping this entry) the application should print the stack-trace, log this event, make the user aware but continue providing its services to support the security of the network.

For example, in golang this can be achieved by catching panic conditions using the Defer, Panic, Recover pattern.

6 Findings

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

6.1 RocketDaoNodeTrusted - DAO takeover during deployment/bootstrapping Critical

Resolution

The node registration is enabled by default (`node.registration.enabled`) but the client intends to change this to disabling the registration until bootstrap mode finished.

> We are intending to set node registrations to false during deployment, then open it up when we need to register our oDAO nodes

Description

The initial deployer of the `RocketStorage` contract is set as the `Guardian` /Bootstrapping role. This guardian can bootstrap the TrustedNode and Protocol DAO, add members, upgrade components, change settings.

Right after deploying the DAO contract the member count is zero. The Guardian can now begin calling any of the bootstrapping functions to add members, change settings, upgrade components, interact with the treasury, etc. The bootstrapping configuration by the Guardian is unlikely to all happen within one transaction which might allow other parties to interact with the system while it is being set up.

`RocketDaoNodeTrusted` also implements a recovery mode that allows **any registered node** to invite themselves directly into the DAO without requiring approval from the Guardian or potential other DAO members **as long as the total member count is below** `daoMemberMinCount` **(3)**. The Guardian itself is not counted as a DAO member as it is a supervisory role.

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/node/RocketDAONodeTrusted.sol:L202-L215

```
/**** Recovery ***************/

// In an explicable black swan scenario where the DAO loses more th
// Must have their ID, email, current RPL bond amount available and
function memberJoinRequired(string memory _id, string memory _ema
    // Ok good to go, lets add them
    (bool successPropose, bytes memory responsePropose) = getCont
    // Was there an error?
    require(successPropose, getRevertMsg(responsePropose));
    // Get the to automatically join as a member (by a regular prop
    (bool successJoin, bytes memory responseJoin) = getContractAd
    // Was there an error?
    require(successJoin, getRevertMsg(responseJoin));
}
```

This opens up a window during the bootstrapping phase where any Ethereum Address might be able to register as a node (`RocketNodeManager.registerNode`) if node registration is enabled (default= `true`) rushing into `RocketDAONodeTrusted.memberJoinRequired` adding themselves (up to 3 nodes) as trusted nodes to the DAO. The new DAO members can now take over the DAO by issuing proposals, waiting 2 blocks to vote/execute them (upgrade, change settings while Guardian is changing settings, etc.). The Guardian role can kick the new DAO members, however, they can invite themselves back into the DAO.

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/protocol/settings/RocketDAOProtocolSettingsNode.sol:L19-L19

```
setSettingBool("node.registration.enabled", true);
```

Recommendation

Disable the DAO recovery mode during bootstrapping. Disable node registration by default and require the guardian to enable it. Ensure that `bootstrapDisable` (in both DAO contracts) performs sanity checks as to whether the DAO bootstrapping finished and permissions can effectively be revoked without putting the DAO at risk or in an irrecoverable state (enough members bootstrapped, vital configurations like registration and other settings are configured, ...).

6.2 RocketTokenRETH - sandwiching opportunity on price updates Critical ✓ Fixed

Resolution

This issue is being addressed in a currently pending pull request. By introducing a delay between an rETH deposit and a subsequent transfer or burn, sandwiching a price update transaction is not possible anymore. Specifically, a deposit delay of circa one day is introduced:

https://github.com/rocket-pool/rocketpool/pull/201/files#diff-0387338dc5dd7edd0a03766cfdaaee42d021d4e781239d5ebbff359c81497839R146-R150

```
    // This is called by the base ERC20 contract before al
    function _beforeTokenTransfer(address from, address,
        // Don't run check if this is a mint transaction
        if (from != address(0)) {
            // Check which block the user's last deposit
            bytes32 key = keccak256(abi.encodePacked("us
            uint256 lastDepositBlock = getUint(key);
            if (lastDepositBlock > 0) {
                // Ensure enough blocks have passed
                RocketDAOProtocolSettingsNetworkInterfac
                uint256 blocksPassed = block.number.sub(
                require(blocksPassed > rocketDAOProtocol
                // Clear the state as it's no longer nece
                deleteUint(key);
            }
        }
    }
```

In the current version, it is correctly enforced that a deposit delay of zero is not possible.

Description

The `rETH` token price is not coupled to the amount of `rETH` tokens in circulation on the Ethereum chain. The price is reported by oracle nodes and committed to the system via a voting process. The price of `rETH` changes If 51% of nodes observe and submit the same price information. If nodes fail to find price consensus for a block, then the `rETH` price might be stale.

There is an opportunity for the user to front-run the price update right before it is committed. If the next price is higher than the previous (typical case), this gives an instant opportunity to perform a risk-free `ETH -> rETH -> ETH` exchange for profit. In the worst case, one could drain all the `ETH` held by the `RocketTokenRETH` contract + excess funds stored in the vault.

Note: there seems to be a `"network.submit.balances.frequency"` price and balance submission frequency of 24hrs. However, this frequency is not enforced, and it is questionable if it makes sense to pin the price for 24hrs.

Note: the total supply of the `RocketTokenRETH` contract may be completely disconnected from the reported total supply for `RETH` via oracle nodes.

Examples

A user observes a price update for `rETH` submitted to `RocketNetworkPrices`, resulting in an increased price for `rETH`. The user front-runs the effective price update (51% consensus reached on submission) by `rETH` at the current, discounted rate. Ideally, the user checks that none of the funds will be assigned to minipools in the queue (empty queue, disabled assignment, ..) and that the expected amount of ETH returned is available `RocketTokenRETH` and `RocketDeposit` (excess funds) contracts. The user then waits for the price update and back-runs it with a call burning all the `rETH` obtained at a discount for `ETH`, realizing an immediate profit.

The amount of ETH was only staked during this one process for the price update duration and unlikely to be useful to the system. This way, a whale (only limited by the max deposit amount set on deposit) can drain the `RocketTokenRETH` contract from all its `ETH` and excess eth funds.

mempool observed: `submitPrice` tx (an effective transaction that changes the price) wrapped with buying `rETH` and selling `rETH` for `ETH`:

- RocketDepositPool.deposit() at old price => mints `rETH` at current rate
- RocketNetworkPrices.submitPrices(newRate)
- RocketTokenRETH.burn(balanceOf(msg.sender) => burns `rETH` for `ETH` at new rate

- deposit (virtually no limit with 1000ETH being the limit right now)

rocketpool-2.5-Tokenomics-updates/contracts/contract/deposit/RocketDepositPool.sol:L63-L67

```
require(rocketDAOProtocolSettingsDeposit.getDepositEnabled(), "De
require(msg.value >= rocketDAOProtocolSettingsDeposit.getMinimumD
require(getBalance().add(msg.value) <= rocketDAOProtocolSettingsD
// Mint rETH to user account
rocketTokenRETH.mint(msg.value, msg.sender);
```

- trustedNodes submitPrice (changes params for `getEthValue` and `getRethValue`)

rocketpool-2.5-Tokenomics-updates/contracts/contract/network/RocketNetworkPrices.sol:L69-L72

```
RocketDAONodeTrustedInterface rocketDAONodeTrusted = RocketDAONod
if (calcBase.mul(submissionCount).div(rocketDAONodeTrusted.getMem
    updatePrices(_block, _rplPrice);
}
```

- immediately burn at new rate (as params for `getEthValue` changed)

rocketpool-2.5-Tokenomics-updates/contracts/contract/token/RocketTokenRETH.sol:L107-L124

```
function burn(uint256 _rethAmount) override external {
    // Check rETH amount
    require(_rethAmount > 0, "Invalid token burn amount");
    require(balanceOf(msg.sender) >= _rethAmount, "Insufficient r
    // Get ETH amount
    uint256 ethAmount = getEthValue(_rethAmount);
    // Get & check ETH balance
    uint256 ethBalance = getTotalCollateral();
    require(ethBalance >= ethAmount, "Insufficient ETH balance fo
    // Update balance & supply
    _burn(msg.sender, _rethAmount);
    // Withdraw ETH from deposit pool if required
    withdrawDepositCollateral(ethAmount);
    // Transfer ETH to sender
    msg.sender.transfer(ethAmount);
    // Emit tokens burned event
    emit TokensBurned(msg.sender, _rethAmount, ethAmount, block.t
}
```

6.3 RocketDaoNodeTrustedActions - Incomplete implementation of member challenge process

Critical ✓ Fixed

Resolution

As of the Smartnode's `v1.0.0-rc1` release, its watchtower process supports challenge detection and responses. It should be noted that a node's challenge state is extracted from the respective storage key. This means that no process currently makes use of the emitted `ActionChallengeMade` event. It can potentially be removed to optimize gas cost.

Description

Any registered (even untrusted) node can challenge a trusted DAO node to respond. The challenge is initiated by calling `actionChallengeMake`. Trusted nodes can challenge for free, other nodes have to provide `members.challenge.cost` as a tribute to the Ethereum gods. The challenged node must call `actionChallengeDecide` before `challengeStartBlock + members.challenge.window` blocks are over (default approx 7 days). However, the Golang codebase does not actively monitor for the `ActionChallengeMade` event, nor does the node - regularly - check if it is being challenged. Means to respond to the challenge (calling `actionChallengeDecide` to stop the challenge) are not implemented.

- Nodes do not seem to monitor `ActionChallengeMade` events so that they could react to challenges
- Nodes do not implement `actionChallengeDecide` and, therefore, cannot successfully stop a challenge
- Funds/Tribute sent along with the challenge will be locked forever in the `RocketDAONodeTrustedActions` contract. There's no means to recover the funds.
- It is questionable whether the incentives are aligned well enough for anyone to challenge stale nodes. The default of `1 eth` compared to the risk of the "malicious" or "stale" node exiting themselves is quite high. The challenger is not incentivized to challenge someone other than for taking over the DAO. If the tribute is too low, this might incentivize users to grief trusted nodes and force them to close a challenge.
- Requiring that the challenge initiator is a different registered node than the challenge finalized is a weak protection since the system is open to anyone to register as a node (even without depositing any funds.)
- block time is subject to fluctuations. With the default of `43204` blocks, the challenge might expire at `5 days` (`10 seconds` block time), `6.5 days` (`13 seconds` **Ethereum target median block time**), `7 days` (`14 seconds`), or more with historic block times going up to `20 seconds` for shorter periods.

A minority of trusted nodes may use this functionality to boot other trusted node members off the DAO issuing challenges once a day until the DAO member number is low enough to allow them to reach quorum for their own proposals or until the member threshold allows them to add new nodes without having to go through the proposal process at all.

Examples

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/node/settings/RocketDAONodeTrustedSettingsMembers.sol:L22-L24

```
setSettingUint('members.challenge.cooldown', 6172);
setSettingUint('members.challenge.window', 43204);
setSettingUint('members.challenge.cost', 1 ether);
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/node/RocketDAONodeTrustedActions.sol:L204-L206

```
// In the event that the majority/all of members go offline permane
// If it does not respond in the given window, it can be removed as
// This should only be used in an emergency situation to recover
```

Recommendation

Implement the challenge-response process before enabling users to challenge other nodes. Implement means to detect misuse of this feature for griefing e.g. when one trusted node member forces another trusted node to defeat challenges over and over again (technical controls, monitoring).

6.4 RocketDAOProtocolSettings/RocketDAONodeTrustedSettings - anyone can set/overwrite settings until contract is declared “deployed” Critical

Acknowledged

Resolution

The client is aware of and acknowledges this potential issue. As with the current contracts the `deployed` flag is always set in the constructor and there will be no window for someone else to interact with the contract before this flag is set. The following statement was provided:

> [...] this method is purely to set the initial default vars. It shouldn’t be run again due to the deployment flag being flagged incase that contract is upgraded and those default vars aren’t removed.

Additionally, it was suggested to add safeguards to the access restricting modifier, to only allowing the guardian to change settings if a settings contract “forgets” to set the `deployed` flag in the constructor (Note: the `deployed` flag must be set with the deploing transaction or else there might be a window for someone to interact with the contract before it is fully configured).

Description

The `onlyDAOProtocolProposal` modifier guards all state-changing methods in this contract. However, analog to issue 6.5, the access control is disabled until the variable `settingsNameSpace.deployed` is set. If this contract is not deployed and configured in one transaction, anyone can update the contract while left unprotected on the blockchain.

See issue 6.5 for a similar issue.

Examples

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/protocol/settings/RocketDAOProtocolSettings.sol:L18-L23

```
modifier onlyDAOProtocolProposal() {
    // If this contract has been initialised, only allow access fro
    if(getBool(keccak256(abi.encodePacked(settingNameSpace, "depl
    _;
}
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/node/settings/RocketDAONodeTrustedSettings.sol:L18-L22

```
modifier onlyDAONodeTrustedProposal() {
    // If this contract has been initialised, only allow access fro
    if(getBool(keccak256(abi.encodePacked(settingNameSpace, "depl
    _;
}
```

There are at least 9 more occurrences of this pattern.

Recommendation

Restrict access to the methods to a temporary trusted account (e.g. `guardian`) until the system bootstrapping phase ends by setting `deployed` to `true`.

6.5 RocketStorage - anyone can set/update values before the contract is initialized Critical ✓ Fixed

Resolution

Fixed by restricting access to the guardian while the contract is not yet `initialized`. The relevant changeset is rocket-pool/rocketpool@`495a51f`. The client provided the following statement:

> tx.origin is only used in this deployment instance and should be safe since no external contracts are

interacted with

The client is aware of the implication of using `tx.origin` and that the guardian should never be used to interact with third-party contracts as the contract may be able to impersonate the guardian changing settings in the storage contract during that transaction.

https://github.com/ConsenSys/rocketpool-audit-2021-03/blob/0a5f680ae0f4da0c5639a241bd1605512cba6004/rocketpool-rp3.0-updates/contracts/contract/RocketStorage.sol#L31-L32

Description

According to the deployment script, the contract is deployed, and settings are configured in multiple transactions. This also means that for a period of time, the contract is left unprotected on the blockchain. Anyone can delete/set any value in the centralized data store. An attacker might monitor the mempool for new deployments of the `RocketStorage` contract and front-run calls to `contract.storage.initialised` setting arbitrary values in the system.

Examples

rocketpool-2.5-Tokenomics-updates/contracts/contract/RocketStorage.sol:L24-L31

```
modifier onlyLatestRocketNetworkContract() {
    // The owner and other contracts are only allowed to set the st
    if (boolStorage[keccak256(abi.encodePacked("contract.storage.
        // Make sure the access is permitted to only contracts in c
        require(boolStorage[keccak256(abi.encodePacked("contract.
    }
    _;
}
```

Recommendation

Restrict access to the methods to a temporary trusted account (e.g. `guardian`) until the system bootstrapping phase ends by setting `initialised` to `true`.

6.6 RocketDAOProposals - Unpredictable behavior due to short vote delay Major ✓ Addressed

Resolution

Addressed in branch `rp3.0-updates` (rocket-pool/rocketpool@`b424ca1`) by changing the default delay `proposal.vote.delay.blocks` to one week.

Description

A proposal can be voted and passed when it enters the `ACTIVE` state. Voting starts when the current `block.number` is greater than the `startBlock` configured in the proposal (up until the `endBlock`). The requirement for the `startBlock` is to be at least greater than `block.number` when the proposal is submitted.

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/RocketDAOProposal.sol:L167-L170

```
require(_startBlock > block.number, "Proposal start block must be
require(_durationBlocks > 0, "Proposal cannot have a duration of
require(_expiresBlocks > 0, "Proposal cannot have a execution exp
require(_votesRequired > 0, "Proposal cannot have a 0 votes requi
```

The default vote delay configured in the system is `1` block.

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/node/settings/RocketDAONodeTrustedSettingsProposals.sol:L21-L21

```
setSettingUint('proposal.vote.delay.blocks', 1);
```

A vote is immediately passed when the required quorum is reached which allows it to be executed. This means that a group that is holding enough voting power can propose a change, wait for two blocks (`block.number (of time of proposal creation) + configuredDelay (1) + 1 (for ACTIVE state)`, then vote and execute for the proposal to pass for it to take effect almost immediately after only 2 blocks (<30seconds).

Settings can be changed after 30 seconds which might be unpredictable for other DAO members and not give them enough time to oppose and leave the DAO.

Recommendation

The underlying issue is that users of the system can't be sure what the behavior of a function call will be, and this is because the behavior can change after two blocks. The only guarantee is that users can be sure the settings don't change for the next block if no proposal is active.

We recommend giving the user advance notice of changes with a delay. For example, all upgrades should require two steps with a mandatory time window between them. The first step merely broadcasts to users that a particular change is coming, and the second step commits that change after a suitable waiting period.

6.7 RocketRewardPool - Unpredictable staking rewards as stake can be added just before claiming and rewards may be paid to to operators that do not provide a service to the system Major

Partially Addressed

Resolution

Partially addressed in branch `rp3.0-updates` (rocket-pool/rocketpool@`b424ca1`) by changing the withdrawal requirements to `150%` of the effective RPL.

The client provided the following statement:

> Node operators can now only withdraw RPL above their 150% effective RPL stake.

Description

Nodes/TrustedNodes earn rewards based on the **current** share of the effective RPL stake provided backing the number of Minipools they run. The reward is paid out regardless of when the effective node stake was provided, as long as it is present just before the call to `claim()`. This means the reward does not take into account how long the stake was provided. The effective RPL stake is the nodes RPL stake capped at a maximum of `halfDepositUserAmount * 150% * nr_of_minipools(node) / RPLPrice`. If the node does not run any Minipools, the effective RPL stake is zero.

Since effective stake can be added just before calling the `claim()` method (effectively trying to get a reward for a period that passed without RPL being staked for the full duration), this might create an unpredictable outcome for other participants, as adding significant stake (requires creating Minipools and staking the max per pool; the stake is locked for at least the duration of a reward period `rpl.rewards.claim.period.blocks`) shifts the shares users get for the fixed total amount of rewards. This can be unfair if the first users claimed their reward, and then someone is artificially inflating the total amount of shares by adding more stake to get a bigger part of the remaining reward. However, this comes at the cost of the registered node having to create more Minipools to stake more, requiring an initial deposit (16ETH, or 0ETH under certain circumstances for trusted nodes) by the actor attempting to get a larger share of the rewards. The risk of losing funds for this actor, however, is rather low, as they can immediately `dissolve()` and `close()` the Minipool to refund their node deposit as `NETH` right after claiming the reward only losing the gas spent on the various transactions.

This can be extended to a node operator creating a Minipool and staking the maximum amount before calling `claim` to remove the Minipool right after, freeing up the `ETH` that was locked in the Minipool until the next reward period starts. The node operator is not providing any service to the network, loses some value in ETH for gas but may compensate that with the RPL staking rewards. If the node amassed a significant amount of RPL stake, they might even try to flash-loan enough ETH to spawn Minipools to inflate

their effective stake and earn most of the rewards to return the loan RPL profit.

```
-- reward period ends -- front-run other claimers to maximize profits
[create x minipools]
[stake to max effective RPL for amount of minipools; locked for 14 days]
[claim rewards for inflated effective RPL stake]
[dissolve(), close() minipools -> refund NETH]
[burn NETH for ETH]
... wait 14 days
[withdraw stake OR start again creating Minipools, claiming rewards while the
```

By staking just before claiming, the node effectively can earn rewards for 2 reward periods by only staking RPL for the duration of one period (claim the previous period, leave it in for 14 days, claim another period, withdraw).

The stake can be withdrawn at the earliest 14 days after staking. However, it can be added back at any time, and the stake addition takes effect immediately. This allows for optimizing the staking reward as follows (assuming we front-run other claimers to maximize profits and perform all transactions in one block):

```
[stake max effective amount for the number of minipools]
[claim() to claim the previous period even though we did not provide any stake
[optionally dissolve Minipools unlocking ETH]
-- stake is locked for at least 14 days --
-- 14 days forward - new reward period started --
[claim() the period]
[withdraw() (leaving min pool stake OR everything if we dissolve all the Minip
[lend RPL to other platforms and earn interest]
-- 14 days forward -new reward period started --
[get RPL back from another platform]
[stake & create minipools to inflate effective stake]
[claim()]
[optionally dissolve Minipools to unlock node ETH]
-- stake is locked for at least 14 days --
-- 14 days forward - new reward period started --
[claim() the period]
[withdraw() (leaving min pool stake OR everything if we dissolve all the Minip
[lend RPL to other platforms and earn interest]
...
```

Note that `withdraw()` can be called right at the time the new reward period starts:

rocketpool-2.5-Tokenomics-updates/contracts/contract/node/RocketNodeStaking.sol:L165-L166

```
require(block.number.sub(getNodeRPLStakedBlock(msg.sender)) >= ro
// Get & check node's current RPL stake
```

Examples

- A node may choose to register and stake some RPL to collect rewards but never actually provide registered node duties, e.g., operating a Minipool.
- Node shares for a passed reward epoch are unpredictable as nodes may change their stake (adding) after/before users claim their rewards.

- A node can maximize its rewards by adding stake just before claiming it
- A node can stake to claim rewards, wait 14 days, withdraw, lend on a platform and return the stake in time to claim the next period.

Recommendation

Review the incentive model for the RPL rewards. Consider adjusting it so that nodes that provide a service get a better share of the rewards. Consider accruing rewards for the duration the stake was provided instead of taking a snapshot whenever the node calls `claim()`. Require stake to be locked for > 14 days instead of >=14 days (`withdraw()`) or have users skip the first reward period after staking.

6.8 RocketNodeStaking - Node operators can reduce slashing impact by withdrawing excess staked RPL Major ✓ Fixed

Resolution

The `RocketNodeStaking.withdrawRPL` method now reverts if a node operator attempts to withdraw an RPL amount that results in the leftover RPL stake being smaller than the maximum required stake. This prevents operators from withdrawing excess RPL to avoid the impact of a slashing.

https://github.com/rocket-pool/rocketpool/blob/rp3.0-updates/contracts/contract/node/RocketNodeStaking.sol#L187

Description

Oracle nodes update the Minipools' balance and progress it to the withdrawable state when they observe the minipools stake to become withdrawable. If the observed stakingEndBalance is less than the user deposit for that pool, the node operator is punished for the difference.

rocketpool-2.5-Tokenomics-updates/contracts/contract/minipool/RocketMinipoolStatus.sol: L89-L94

```
rocketMinipoolManager.setMinipoolWithdrawalBalances(_minipoolAddr
// Apply node penalties by liquidating RPL stake
if (_stakingEndBalance < userDepositBalance) {
    RocketNodeStakingInterface rocketNodeStaking = RocketNodeStak
    rocketNodeStaking.slashRPL(minipool.getNodeAddress(), userDep
}
```

The amount slashed is at max `userDepositBalance - stakingEndBalance`. The `userDepositBalance` is at least `16 ETH` (minipool.half/.full) and at max `32 ETH` (minipool.empty). The maximum amount to be slashed is therefore `32 ETH` (endBalance = 0, minipool.empty).

The slashing amount is denoted in `ETH`. The `RPL` price (in `ETH`) is updated regularly by oracle nodes (see related issue issue 6.2; note that the RPL token is potentially affected by a similar issue as one can stake RPL, wait for the cooldown period & wait for the price to change, and withdraw stake at higher RPL price/ETH). The `ETH` amount to be slashed is converted to `RPL`, and the corresponding `RPL` stake is slashed.

rocketpool-2.5-Tokenomics-updates/contracts/contract/node/RocketNodeStaking.sol:L188-L196

```
uint256 rplSlashAmount = calcBase.mul(_ethSlashAmount).div(rocket
// Cap slashed amount to node's RPL stake
uint256 rplStake = getNodeRPLStake(_nodeAddress);
if (rplSlashAmount > rplStake) { rplSlashAmount = rplStake; }
// Transfer slashed amount to auction contract
rocketVault.transferToken("rocketAuctionManager", getContractAddr
// Update RPL stake amounts
decreaseTotalRPLStake(rplSlashAmount);
decreaseNodeRPLStake(_nodeAddress, rplSlashAmount);
```

If the node does not have a sufficient `RPL` stake to cover the losses, the slashing amount is capped at whatever amount of `RPL` the node has left staked.

The minimum amount of `RPL` a node needs to have staked if it operates minipools is calculated as follows:

rocketpool-2.5-Tokenomics-updates/contracts/contract/node/RocketNodeStaking.sol:L115-L120

```
    // Calculate minimum RPL stake
    return rocketDAOProtocolSettingsMinipool.getHalfDepositUserAm
        .mul(rocketDAOProtocolSettingsNode.getMinimumPerMinipoolS
        .mul(rocketMinipoolManager.getNodeMinipoolCount(_nodeAddr
        .div(rocketNetworkPrices.getRPLPrice());
}
```

With the current configuration, this would resolve in a minimum stake of `16 ETH * 0.1 (10% collateralization) * 1 (nr_minipools) * RPL_Price` for a node operating 1 minipool. This means a node operator basically only needs to have 10% of `16 ETH` staked to operate one minipool.

An operator can withdraw their stake at any time, but they have to wait at least 14 days after the last time they staked (cooldown period). They can, at max, withdraw all but the minimum stake required to run the pools (nr_of_minipools * 16 ETH * 10%). This also means that after the cooldown period, they can reduce their stake to 10% of the half deposit amount (16ETH), then perform a

voluntary exit on ETH2 so that the minipool becomes `withdrawable`. If they end up with less than the `userDepositBalance` in staking rewards, they would only get slashed the `1.6 ETH` at max (10% of 16ETH half deposit amount for 1 minipool) even though they incurred a loss that may be up to 32 ETH (empty Minipool empty amount).

Furthermore, if a node operator runs multiple minipools, let's say 5, then they would have to provide at least `5*16ETH*0.1 = 8ETH` as a security guarantee in the form of staked RPL. If the node operator incurs a loss with one of their minipools, their 8 ETH RPL stake will likely be slashed in full. Their other - still operating - minipools are not backed by any RPL anymore, and they effectively cannot be slashed anymore. This means that a malicious node operator can create multiple minipools, stake the minimum amount of RPL, get slashed for one minipool, and still operate the others without having the minimum RPL needed to run the minipools staked (getNodeMinipoolLimit).

The RPL stake is donated to the RocketAuctionManager, where they can attempt to buy back RPL potentially at a discount.

Note: Staking more RPL (e.g., to add another Minipool) resets the cooldown period for the total RPL staked (not only for the newly added)

Recommendation

It is recommended to redesign the withdrawal process to prevent users from withdrawing their stake while slashable actions can still occur. A potential solution may be to add a locking period in the process. A node operator may schedule the withdrawal of funds, and after a certain time has passed, may withdraw them. This prevents the immediate withdrawal of funds that may need to be reduced while slashable events can still occur. E.g.:

- A node operator requests to withdraw all but the minimum required stake to run their pools.
- The funds are scheduled for withdrawal and locked until a period of X days has passed.
- (optional) In this period, a slashable event occurs. The funds for compensation are taken from the user's stake including the funds scheduled for withdrawal.
- After the time has passed, the node operator may call a function to trigger the withdrawal and get paid out.

6.9 RocketTokenRPL - inaccurate inflation rate and potential for manipulation lowering the real APY

Major ✓ Addressed

Resolution

The main issue was addressed in branch `rp3.0-updates` (rocket-pool/rocketpool@`b424ca1`) by recording the timestamp up to when inflation was updated to instead of the current block timestamp (`lastTimeOfInflationMint + interval * inflationIntervals`).

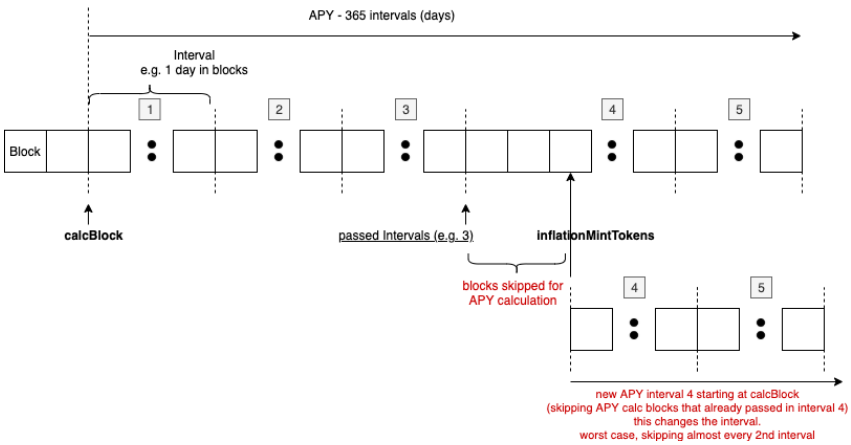
Description

RocketTokenRPL allows users to swap their fixed-rate tokens to the inflationary RocketTokenRPL ERC20 token via a `swapToken` function. The DAO defines the inflation rate of this token and is initially set to be 5% APY. This APY is configured as a daily inflation rate (APD) with the corresponding `1 day in blocks` inflation interval in the `rocketDAOProtocolSettingsInflation` contract. The DAO members control the inflation settings.

Anyone can call `inflationMintTokens` to inflate the token, which mints tokens to the contracts RocketVault. Tokens are minted for discreet intervals since the last time `inflationMintTokens` was called (recorded as `inflationCalcBlock`). The inflation is then calculated for the passed intervals without taking the current not yet completed interval. However, the `inflationCalcBlock` is set to the current `block.number`, effectively skipping some "time"/blocks of the APY calculation.

The more often `inflationMintTokens` is called, the higher the APY likelihood dropping below the configured 5%. In the worst case, one could manipulate the APY down to 2.45% (assuming that the APD for a 5% APY was configured) by calling `inflationMintTokens` close to the end of every second interval. This would essentially restart the APY interval at `block.number`, skipping blocks of the current interval that have not been accounted for.

The following diagram illustrates the skipped blocks due to the incorrect recording of `inflationCalcBlock` as `block.number`. The example assumes that we are in interval 4 but have not completed it. `3` APD intervals have passed, and this is what the inflation rate is based on. However, the `inflationCalcBlock` is updated to the current `block.number`, skipping some time/blocks that are now unaccounted in the APY restarting the 4th interval at `block.number`.



- Note: updating the inflation rate will directly affect past inflation intervals that have not been minted! this might be undesirable, and it could be considered to force an inflation mint if the APY changes
- Note: if the interval is small enough and there is a history of unaccounted intervals to be minted, and the Ethereum network is congested, gas fees may be high and block limits hit, the calculations in the for loop might be susceptible to DoS the inflation mechanism because of gas constraints.
- Note: The inflation seems only to be triggered regularly on `RocketRewardsPool.claim` (or at any point by external actors). If the price establishes based on the total supply of tokens, then this may give attackers an opportunity to front-run other users trading large amounts of RPL that may previously have calculated their prices based on the un-inflated supply.
- Note: that the discrete interval-based inflation (e.g., once a day) might create dynamics that put pressure on users to trade their RPL in windows instead of consecutively

Examples

- the inflation intervals passed is the number of completed intervals. The current interval that is started is not included.

rocketpool-2.5-Tokenomics-updates/contracts/contract/token/RocketTokenRPL.sol:L108-L119

```
function getInlfationIntervalsPassed() override public view retur
    // The block that inflation was last calculated at
    uint256 inflationLastCalculatedBlock = getInflationCalcBlock(
    // Get the daily inflation in blocks
    uint256 inflationInterval = getInflationIntervalBlocks();
    // Calculate now if inflation has begun
    if(inflationLastCalculatedBlock > 0) {
        return (block.number).sub(inflationLastCalculatedBlock).d
    }else{
        return 0;
    }
}
```

- the inflation calculation calculates the to-be-minted tokens for the inflation rate at `newTokens = supply * rateAPD^intervals - supply`

rocketpool-2.5-Tokenomics-updates/contracts/contract/token/RocketTokenRPL.sol:L126-L148

```
function inflationCalculate() override public view returns (uint2
    // The inflation amount
    uint256 inflationTokenAmount = 0;
    // Optimisation
    uint256 inflationRate = getInflationIntervalRate();
    // Compute the number of inflation intervals elapsed since the
    uint256 intervalsSinceLastMint = getInlfationIntervalsPassed(
    // Only update  if last interval has passed and inflation rate
    if(intervalsSinceLastMint > 0 && inflationRate > 0) {
        // Our inflation rate
        uint256 rate = inflationRate;
        // Compute inflation for total inflation intervals elapsed
        for (uint256 i = 1; i < intervalsSinceLastMint; i++) {
            rate = rate.mul(inflationRate).div(10 ** 18);
        }
        // Get the total supply now
        uint256 totalSupplyCurrent = totalSupply();
        // Return inflation amount
        inflationTokenAmount = totalSupplyCurrent.mul(rate).div(1
    }
    // Done
    return inflationTokenAmount;
}
```

Recommendation

Properly track `inflationCalcBlock` as the end of the previous interval, as this is up to where the inflation was calculated, instead of the block at which the method was invoked.

Ensure APY/APD and interval configuration match up. Ensure the interval is not too small (potential gas DoS blocking inflation mint and `RocketRewardsPool.claim`).

6.10 Trusted node participation risk and potential client “optimizations” Major ✓ Fixed

Resolution

The development team considers this issue fixed as monitoring on the correct behaviour of node software is added to the system.

Description

The system might end up in a stale state with minipools never being `setWithdrawable` or network and prices being severely outdated because trusted nodes don’t fulfill their duty of providing oracle values. Minipools not being able to advance to the `Withdrawable` state will severely harm the system as no rewards can be paid out. Outdated balances and prices may affect token economics around the tokens involved (specifically `rETH` price depends on oracle observations).

There is an incentive to be an oracle node as you get paid to provide oracle node duties when enrolled with the DAO. However, it is not enforced that nodes actually fulfill their duty of calling the

respective `onlyTrustedNode` oracle functions to submit prices/balances/minipool rewards.

Therefore, a smart Rocket Pool trusted node operator might consider patching their client software to not or only sporadically fulfill their duties to save considerable amounts of gas, making more profit than other trusted nodes would.

There is no means to directly incentivize trusted nodes to call certain functions as they get their rewards anyway. The only risk they run is that other trusted nodes might detect their antisocial behavior and attempt to kick them out of the DAO. To detect this, monitoring tools and processes need to be established; it is questionable whether users would participate in high maintenance DAO operators.

Furthermore, trusted nodes might choose to gas optimize their submissions to avoid calling the actual action once quorum was established. They can, for example, attempt to submit prices as early as possible, avoiding that they're the first to hit the 51% threshold.

Recommendation

Create monitoring tools and processes to detect participants that do not fulfill their trusted DAO duties. Create direct incentives for trusted nodes to provide oracle services by, e.g., recording their participation rate and only payout rewards based on how active they are.

6.11 RocketDAONodeTrustedUpgrade - upgrade does not prevent the use of the same address multiple times creating an inconsistency where `getContractAddress` returns outdated information Major ✓ Fixed

Resolution

A check has been introduced to make sure that the new contract address is not already in use by checking against the corresponding `contract.exists` storage key.

Description

When adding a new contract, it is checked whether the address is already in use. This check is missing when upgrading a named contract to a new implementation, potentially allowing someone to register one address to multiple names creating an inconsistent configuration.

The crux of this is, that, `getContractAddress()` will now return a contract address that is not registered anymore (while

`getContractName` may throw). `getContractAddress` can therefore not relied upon when checking ACL.

- **add** contract `name=test, address=0xfefe` ->

```
sets contract.exists.0xfefe=true
sets contract.name.0xfefe=test
sets contract.address.test=0xfefe
sets contract.abi.test=abi
```

- **add** another contract `name=badcontract, address=0xbadbad` ->

```
sets contract.exists.0xbadbad=true
sets contract.name.0xbadbad=badcontract
sets contract.address.badcontract=0xbadbad
sets contract.abi.badcontract=abi
```

- **update** contract `name=test, address=0xbadbad` reusing badcontradcts address, the address is now bound to 2 names (test, badcontract)

```
overwrites contract.exists.0xbadbad=true` (even though its already true)
updates contract.name.0xbadbad=test (overwrites the reference to badcontr
updates contract.address.test=0xbadbad (ok, expected)
updates contract.abi.test=abi (ok, expected)
removes contract.name.0xfefe (ok)
removes contract.exists.0xfefe (ok)
```

- update contract `name=test, address=0xc0c0`

```
sets contract.exists.0xc0c0=true
sets contract.name.0xc0c0=test (ok, expected)
updates contract.address.test=0xc0c0 (ok, expected)
updates contract.abi.test=abi (ok, expected)
removes contract.name.0xbadbad (the contract is still registered as badcc
removes contract.exists.0xbadbad (the contract is still registered as bac
```

After this, `badcontract` is partially cleared, `getContractName(0xbadbad)` throws while `getContractAddress(badcontract)` returns `0xbadbad` which is already unregistered (`contract.exists.0xbadbad=false`)

```
(removed) contract.exists.0xbadbad
(removed) contract.name.0xbadbad=badcontract
sets contract.address.badcontract=0xbadbad
sets contract.abi.badcontract=abi
```

Examples

- check in `_addContract`\`

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/node/RocketDAONodeTrustedUpgrade.sol:L76-L76

```
require(_contractAddress != address(0x0), "Invalid contract addre
```

- no checks in `upgrade.`

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/node/RocketDAONodeTrustedUpgrade.sol:L53-L59

```
require(_contractAddress != address(0x0), "Invalid contract addre
require(_contractAddress != oldContractAddress, "The contract add
// Register new contract
setBool(keccak256(abi.encodePacked("contract.exists", _contractAd
setString(keccak256(abi.encodePacked("contract.name", _contractAd
setAddress(keccak256(abi.encodePacked("contract.address", _name))
setString(keccak256(abi.encodePacked("contract.abi", _name)), _co
```

Recommendation

Check that the address being upgraded to is not yet registered and properly clean up `contract.address.<name|abi>`.

6.12 Rocketpool CLI - Lax data validation and output sanitation Major ✓ Addressed

Resolution

Addressed with v1.0.0-rc1 by sanitizing non-printables from strings stored in the smart contract.

This effectively mitigates terminal-based control character injection attacks. However, might still be used to inject context-sensitive information that may be consumed by different protocols/presentation layers (web, terminal by displaying falsified information next to fields).

E-mail and timezone format validation was introduced with https://github.com/rocket-pool/rocketpool-go/blob/c8738633ab973503b79c7dee5c2f78d7e44e48ae/dao/trustednode/proposals.go#L22 and rocket-pool/rocketpool-go@`6e72501`.

It is recommended to further tighten the checks on untrusted information enforcing an expected format of information and reject to interact with nodes/data that does not comply with the expected formats (e.g. email being in an email format, timezone information is a valid timezone, and does not contain extra information, ...).

Description

`ValidateTimezoneLocation` and `ValidateDAOMemberEmail` are only used to validate user input from the command line. Timezone location information and member email addresses are stored in the smart contract's string storage, e.g., using the `setTimezoneLocation` function of the `RocketNodeManager` contract. This function only validates that a minimum length of 4 has been given.

Through direct interaction with the contract, an attacker can submit arbitrary information, which is not validated on the CLI's side. With additional integrations of the Rocketpool smart

contracts, the timezone location field may be used by an attacker to inject malicious code (e.g., for cross-site scripting attacks) or injecting false information (e.g. `Balance: 1000 RPL` or `Status: Trusted`), which is directly displayed on a user-facing application.

On the command line, control characters such as newline characters can be injected to alter how text is presented to the user, effectively exploiting user trust in the official application.

Examples

rocketpool-go-2.5-Tokenomics/node/node.go:L134-L153

```
    wg.Go(func() error {
        var err error
        timezoneLocation, err = GetNodeTimezoneLocation(rp, nodeA
        return err
    })

    // Wait for data
    if err := wg.Wait(); err != nil {
        return NodeDetails{}, err
    }

    // Return
    return NodeDetails{
        Address: nodeAddress,
        Exists: exists,
        WithdrawalAddress: withdrawalAddress,
        TimezoneLocation: timezoneLocation,
    }, nil

}
```

smartnode-2.5-Tokenomics/rocketpool-cli/odao/members.go:L34-L44

```
for _, member := range members.Members {
    fmt.Printf("--------------------\n")
    fmt.Printf("\n")
    fmt.Printf("Member ID:            %s\n", member.ID)
    fmt.Printf("Email address:        %s\n", member.Email)
    fmt.Printf("Joined at block:      %d\n", member.JoinedBlock)
    fmt.Printf("Last proposal block:  %d\n", member.LastProposalB
    fmt.Printf("RPL bond amount:      %.6f\n", math.RoundDown(eth
    fmt.Printf("Unbonded minipools:   %d\n", member.UnbondedValid
    fmt.Printf("\n")
}
```

Recommendation

Validate user input before storing it on the blockchain. Validate and sanitize stored user tainted data before presenting it. Establish a register of data validation rules (e.g., email format, timezone format, etc.). Reject nodes operating with nodes that do not honor data validation rules.

Validate the correct format of variables (e.g., timezone location, email, name, ...) on the storage level (if applicable) and the lowest level of the go library to offer developers a strong foundation to

build on and mitigate the risk in future integrations. Furthermore, on-chain validation might not be implemented (due to increased gas consumption) should be mentioned in the developer documentation security section as they need to be handled with special caution by consumer applications. Sanitize output before presenting it to avoid control character injections in terminal applications or other presentation technologies (e.g., SQL or HTML).

Review all usage of the `fmt` lib (especially `Sprintf` and string handling/concatenating functions). Ensure only sanitized data can reach this sink. Review the logging library and ensure it is hardened against control character injection by encoding non-printables and CR-LF.

6.13 Rocketpool CLI - Various command injection vectors Major ✓ Addressed

Resolution

Initially, the client implemented the suggested fix using `%q` to dblquot user-provided data. While this recommendation mitigates some vectors it might still be susceptible to command injection attacks using backticks (as in bash dblquots do not prevent subcommand from being executed - backticks/$(cmd)) and the dblquot sequence may be terminated injecting a `"` which is turned into a `\"`. This was later changed to using golang `shellEscape` in https://github.com/rocket-pool/smartnode/compare/extra-escapes.

Description

Various commands in the Rocketpool CLI make use of the `readOutput` and `printOutput` functions. These do not perform sanitization of user-supplied inputs and allow an attacker to supply malicious values which can be used to execute arbitrary commands on the user's system.

Examples

All commands using the `Client.readOutput`, `Client.printOutput` and `Client.compose` functions are affected.

```
➜  rocketpool-cli git:(master) ✗ ./rocketpool-cli-linux-amd64 service install --yes --network=';head -n3 /etc/passwd'

==> /etc/passwd <==
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
Could not install Rocket Pool service: head: cannot open 'latest' for reading: No such file or directory

➜  rocketpool-cli git:(master) ✗ ./rocketpool-cli-linux-amd64 service install --yes --version=';head -n3 /etc/passwd'

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin

The Rocket Pool service was successfully installed locally!

Please start a new shell session to apply updated user permissions.
(To start a new shell session, log out and back in.)

Run 'rocketpool service config' to configure the service before starting it.
```

Furthermore, `Client.callAPI` is used for API-related calls throughout the Rocketpool service. However, it does not validate that the values passed into it are valid API commands. This can lead to arbitrary command execution, also inside the container using `docker exec`.

Recommendation

Perform strict validation on all user-supplied parameters. If parameter values need to be inserted into a command template string, the `%q` format string or other restrictive equivalents should be used.

6.14 RocketStorage - Risk concentration by giving all registered contracts permissions to change any settings in RocketStorage Major Acknowledged

Resolution

The client provided the following statement:

> We've looked at adding access control contracts using namespaces, but the increase in gas usage would be significant and could hinder upgrades.

Description

The ACL for changing settings in the centralized `RocketStorage` allows any registered contract (listed under `contract.exists`) to change settings that belong to other parts of the system.

The concern is that if someone finds a way to add their malicious contract to the registered contact list, they will override any setting in the system. The storage is authoritative when checking certain ACLs. Being able to set any value might allow an attacker to gain control of the complete system. Allowing any contract to overwrite other contracts' settings dramatically increases the attack surface.

Examples

rocketpool-2.5-Tokenomics-updates/contracts/contract/RocketStorage.sol:L24-L32

```
modifier onlyLatestRocketNetworkContract() {
    // The owner and other contracts are only allowed to set the st
    if (boolStorage[keccak256(abi.encodePacked("contract.storage.
        // Make sure the access is permitted to only contracts in c
        require(boolStorage[keccak256(abi.encodePacked("contract.
    }
    _;
}
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/RocketStorage.sol:L78-L85

```
function setAddress(bytes32 _key, address _value) onlyLatestRocke
    addressStorage[_key] = _value;
}

/// @param _key The key for the record
function setUint(bytes32 _key, uint _value) onlyLatestRocketNetwo
    uIntStorage[_key] = _value;
}
```

Recommendation

Allow contracts to only change settings related to their namespace.

6.15 RocketDAOProposals - require a minimum participation quorum for DAO proposals Medium

✓ Addressed

Resolution

Addressed by requiring the DAO minimum viable user count as the minium quorum with rocket-pool/rocketpool@ `11bc18c` (in bootstrap mode). The check for the bootstrap mode has since been removed following our remark

> [...] the problem here was not so much the bootstrap mode but rather that the dao membership may fall below the recovery mode threshold. The question is, whether it should still be allowed to propose and execute votes if the memberCount at proposal time is below that treshold (e.g. malicious member boots off other members, sends new proposals (quorum required=1), dao membrers rejoin but cannot reject that proposal anymore). Question is if quorum should be at least the recovery treshold.

And the following feedback from the client:

> [...] had that as allowed to happen if bootstrap mode was enabled. I've just disabled the check for bootstrap mode now so that any proposals can't be made if the min member count is below the amount required. This means new members can only be added in this case via the emergency join function before new proposals can be added

Description

If the DAO falls below the minimum viable membership threshold, voting for proposals still continues as DAO proposals do not require a minimum participation quorum. In the worst case, this would allow the last standing DAO member to create a proposal that would be passable with only one vote even if new members would be immediately ready to join via the recovery mode (which has its own risks) as the minimum votes requirement for proposals is set as `>0`.

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/RocketDAOProposal.sol:L170-L170

```
require(_votesRequired > 0, "Proposal cannot have a 0 votes requi
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/node/RocketDAONodeTrustedProposals.sol:L57-L69

```
function propose(string memory _proposalMessage, bytes memory _pa
    // Load contracts
    RocketDAOProposalInterface daoProposal = RocketDAOProposalInt
    RocketDAONodeTrustedInterface daoNodeTrusted = RocketDAONodeT
    RocketDAONodeTrustedSettingsProposalsInterface rocketDAONodeT
    // Check this user can make a proposal now
    require(daoNodeTrusted.getMemberLastProposalBlock(msg.sender)
    // Record the last time this user made a proposal
    setUint(keccak256(abi.encodePacked(daoNameSpace, "member.prop
    // Create the proposal
    return daoProposal.add(msg.sender, 'rocketDAONodeTrustedPropo
}
```

Sidenote: Since a proposals acceptance quorum is recorded on proposal creation, this may lead to another scenario where proposals acceptance quorum may never be reached if members leave the DAO. This would require a re-submission of the proposal.

Recommendation

Do not accept proposals if the member count falls below the minimum DAO membercount threshold.

6.16 RocketDAONodeTrustedUpgrade - inconsistent upgrade blacklist Medium ✓ Addressed

Resolution

Addressed in branch `rp3.0-updates` (rocket-pool/rocketpool@`b424ca1`) by updating the blacklist.

Description

`upgradeContract` defines a hardcoded list of contracts that cannot be upgraded because they manage their own settings (statevars) or

they hold value in the system.

- the list is hardcoded and cannot be extended when new contracts are added via `addcontract`. E.g. what if another contract holding value is added to the system? This would require an upgrade of the upgrade contract to update the whitelist (gas hungry, significant risk of losing access to the upgrade mechanisms if a bug is being introduced).
- a contract named `rocketPoolToken` is blacklisted from being upgradeable but the system registers no contract called `rocketPoolToken`. This may be an oversight or artifact of a previous iteration of the code. However, it may allow a malicious group of nodes to add a contract that is not yet in the system which cannot be removed anymore as there is no `removeContract` functionality and `upgradeContract` to override the malicious contract will fail due to the blacklist.
- Note that upgrading `RocketTokenRPL` requires an account balance migration as contracts in the system may hold value in `RPL` (e.g. a lot in AuctionManager) that may vanish after an upgrade. The contract is not exempt from upgrading. A migration may not be easy to perform as the system cannot be paused to e.g. snapshot balances.

Examples

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/node/RocketDAONodeTrustedUpgrade.sol:L41-L49

```
function _upgradeContract(string memory _name, address _contractA
    // Check contract being upgraded
    bytes32 nameHash = keccak256(abi.encodePacked(_name));
    require(nameHash != keccak256(abi.encodePacked("rocketVault")
    require(nameHash != keccak256(abi.encodePacked("rocketPoolTok
    require(nameHash != keccak256(abi.encodePacked("rocketTokenRE
    require(nameHash != keccak256(abi.encodePacked("rocketTokenNE
    require(nameHash != keccak256(abi.encodePacked("casperDeposit
    // Get old contract address & check contract exists
```

Recommendation

- Consider implementing a whitelist of contracts that are allowed to be upgraded instead of a more error-prone blacklist of contracts that cannot be upgraded.
- Provide documentation that outlines what contracts are upgradeable and why.
- Create a process to verify the blacklist before deploying/operating the system.
- Plan for migration paths when upgrading contracts in the system
- Any proposal that reaches the upgrade contract must be scrutinized for potential malicious activity (e.g. as any registered contract can directly modify storage or may contain subtle backdoors. Upgrading without performing a thorough security inspection may easily put the DAO at risk)

6.17 RocketDAONodeTrustedActions - member cannot be kicked if the vault does not hold enough RPL to cover the bond Medium ✓ Fixed

Resolution

Addressed in branch `rp3.0-updates` (rocket-pool/rocketpool@`b424ca1`) by returning the bond if enough RPL is in the treasury and else continue without returning the bond. This way the member kick action does not block and the member can be kicked regardless of the RPL balance.

Description

If a DAO member behaves badly other DAO members may propose the node be evicted from the DAO. If for some reason, `RocketVault` does not hold enough `RPL` to pay back the DAO member bond `actionKick` will throw. The node is not evicted.

Now this is a somewhat exotic scenario as the vault should always hold the bond for the members in the system. However, if the node was kicked for stealing RPL (e.g. passing an upgrade proposal to perform an attack) it might be impossible to execute the eviction.

Recommendation

Ensure that there is no way a node can influence a succeeded `kick` proposal to fail. Consider burning the bond (by keeping it) as there is a reason for evicting the node or allow them to redeem it in a separate step.

6.18 RocketMinipoolStatus - DAO Membership changes can result in votes getting stuck Medium ✓ Fixed

Resolution

This issue has been fixed in PR https://github.com/ConsenSys/rocketpool-audit-2021-03/issues/204 by introducing a public method that allows anyone to manually trigger a DAO consensus threshold check and a subsequent balance update in case the issue's example scenario occurs.

Description

Changes in the DAO's trusted node members are reflected in the `RocketDAONodeTrusted.getMemberCount()` function. When compared with the vote on consensus threshold, a DAO-driven decision is made,

e.g., when updating token price feeds and changing Minipool states.

Especially in the early phase of the DAO, the functions below can get stuck as execution is restricted to DAO members who have not voted yet. Consider the following scenario:

- The DAO consists of five members
- Two members vote to make a Minipool withdrawable
- The other three members are inactive, the community votes, and they get kicked from the DAO
- The two remaining members have no way to change the Minipool state now. All method calls to trigger the state update fails because the members have already voted before.

Note: votes of members that are kicked/leave are still count towards the quorum!

Examples

Setting a Minipool into the withdrawable state:

rocketpool-2.5-Tokenomics-updates/contracts/contract/minipool/RocketMinipoolStatus.sol:L62-L65

```
RocketDAONodeTrustedInterface rocketDAONodeTrusted = RocketDAONod
if (calcBase.mul(submissionCount).div(rocketDAONodeTrusted.getMem
    setMinipoolWithdrawable(_minipoolAddress, _stakingStartBalanc
}
```

Submitting a block's network balances:

rocketpool-2.5-Tokenomics-updates/contracts/contract/network/RocketNetworkBalances.sol:L94-L97

```
RocketDAONodeTrustedInterface rocketDAONodeTrusted = RocketDAONod
if (calcBase.mul(submissionCount).div(rocketDAONodeTrusted.getMem
    updateBalances(_block, _totalEth, _stakingEth, _rethSupply);
}
```

Submitting a block's RPL price information:

rocketpool-2.5-Tokenomics-updates/contracts/contract/network/RocketNetworkPrices.sol:L69-L72

```
RocketDAONodeTrustedInterface rocketDAONodeTrusted = RocketDAONod
if (calcBase.mul(submissionCount).div(rocketDAONodeTrusted.getMem
    updatePrices(_block, _rplPrice);
}
```

Recommendation

The conditional check and update of price feed information, Minipool state transition, etc., should be externalized into a separate public function. This function is also called internally in the existing code. In case the DAO gets into the scenario above, anyone can call the function to trigger a reevaluation of the condition with updated membership numbers and thus get the process unstuck.

6.19 Trusted/Oracle-Nodes can vote multiple times for different outcomes Medium

Description

Trusted/oracle nodes submit various ETH2 observations to the RocketPool contracts. When 51% of nodes submitted the same observation, the result is stored in the contract. However, while it is recorded that a node already voted for a specific minipool (being withdrawable & balance) or block (price/balance), a re-submission with different parameters for the same minipool/block is not rejected.

Since the oracle values should be distinct, clear, and there can only be one valid value, it should not be allowed for trusted nodes to change their mind voting for multiple different outcomes within one block or one minipool

Examples

- `RocketMinipoolStatus` - a trusted node can submit multiple different results for one minipool

Note that

```
setBool(keccak256(abi.encodePacked("minipool.withdrawable.submitted.node",
msg.sender, _minipoolAddress)), true);
```

is recorded but never checked. (as for the other two instances)

rocketpool-2.5-Tokenomics-updates/contracts/contract/minipool/RocketMinipoolStatus.sol: L48-L57

```
// Get submission keys
bytes32 nodeSubmissionKey = keccak256(abi.encodePacked("minipool.
bytes32 submissionCountKey = keccak256(abi.encodePacked("minipool
// Check & update node submission status
require(!getBool(nodeSubmissionKey), "Duplicate submission from n
setBool(nodeSubmissionKey, true);
setBool(keccak256(abi.encodePacked("minipool.withdrawable.submitt
// Increment submission count
uint256 submissionCount = getUint(submissionCountKey).add(1);
setUint(submissionCountKey, submissionCount);
```

- `RocketNetworkBalances` - a trusted node can submit multiple different results for the balances at a specific block

rocketpool-2.5-Tokenomics-updates/contracts/contract/network/RocketNetworkBalances.sol:L80-L92

```
// Get submission keys
bytes32 nodeSubmissionKey = keccak256(abi.encodePacked("network.b
bytes32 submissionCountKey = keccak256(abi.encodePacked("network.
// Check & update node submission status
require(!getBool(nodeSubmissionKey), "Duplicate submission from n
setBool(nodeSubmissionKey, true);
setBool(keccak256(abi.encodePacked("network.balances.submitted.no
// Increment submission count
uint256 submissionCount = getUint(submissionCountKey).add(1);
setUint(submissionCountKey, submissionCount);
// Emit balances submitted event
emit BalancesSubmitted(msg.sender, _block, _totalEth, _stakingEth
// Check submission count & update network balances
```

- `RocketNetworkPrices` - a trusted node can submit multiple different results for the price at a specific block

rocketpool-2.5-Tokenomics-updates/contracts/contract/network/RocketNetworkPrices.sol:L55-L67

```
// Get submission keys
bytes32 nodeSubmissionKey = keccak256(abi.encodePacked("network.p
bytes32 submissionCountKey = keccak256(abi.encodePacked("network.
// Check & update node submission status
require(!getBool(nodeSubmissionKey), "Duplicate submission from n
setBool(nodeSubmissionKey, true);
setBool(keccak256(abi.encodePacked("network.prices.submitted.node
// Increment submission count
uint256 submissionCount = getUint(submissionCountKey).add(1);
setUint(submissionCountKey, submissionCount);
// Emit prices submitted event
emit PricesSubmitted(msg.sender, _block, _rplPrice, block.timesta
// Check submission count & update network prices
```

Recommendation

Only allow one vote per minipool/block. Don't give nodes the possibility to vote multiple times for different outcomes.

6.20 RocketTokenNETH - Pot. discrepancy between minted tokens and deposited collateral

Medium ✓ Fixed

Resolution

This issue is obsoleted by the fact that the `nETH` contract was removed completely. The client provided the following statement:

> nETH has been removed completely.

Description

The `nETH` token is paid to node operators when minipool becomes withdrawable. `nETH` is supposed to be backed by `ETH` 1:1. However, in most cases, this will not be the case.

The `nETH` minting and deposition of collateral happens in two different stages of a minipool. `nETH` is minted in the minipool state transition from `Staking` to `Withdrawable` when the trusted/oracle nodes find consensus on the fact that the minipool became withdrawable (`submitWinipoolWithdrawable`).

rocketpool-2.5-Tokenomics-updates/contracts/contract/minipool/RocketMinipoolStatus.sol: L63-L65

```
if (calcBase.mul(submissionCount).div(rocketDAONodeTrusted.getMem
    setMinipoolWithdrawable(_minipoolAddress, _stakingStartBalanc
}
```

When consensus is found on the state of the minipool, `nETH` tokens are minted to the `minipool` address according to the withdrawal amount observed by the trusted/oracle nodes. At this stage, `ETH` backing the newly minted `nETH` was not yet provided.

rocketpool-2.5-Tokenomics-updates/contracts/contract/minipool/RocketMinipoolStatus.sol: L80-L87

```
uint256 nodeAmount = getMinipoolNodeRewardAmount(
    minipool.getNodeFee(),
    userDepositBalance,
    minipool.getStakingStartBalance(),
    minipool.getStakingEndBalance()
);
// Mint nETH to minipool contract
if (nodeAmount > 0) { rocketTokenNETH.mint(nodeAmount, _minipoolA
```

The `nETH` token contract now holds more `nETH.totalsupply` than actual `ETH` collateral. It is out of sync with the `ETH` reserve and therefore becomes undercollateralized. This should generally be avoided as the security guarantees that for every `nETH` someone deposited, `ETH` does not hold. However, the newly minted `nETH` is locked to the `minipoolAddress`, and the minipool has no means of redeeming the `nETH` for `ETH` directly (via `nETH.burn()`).

The transition from Withdrawable to `Destroyed` the actual collateral for the previously minted `nETH` (still locked to `minipoolAddress`) is provided by the `Eth2` withdrawal contract. There is no specification for the withdrawal contract as of now. Still, it is assumed that some entity triggers the payout for the Eth2 rewards on the withdrawal contract, which sends the amount of `ETH` to the configured withdrawal address (the `minipoolAddress`).

The `minipool.receive()` function receives the `ETH`

rocketpool-2.5-Tokenomics-updates/contracts/contract/minipool/RocketMinipool.sol:L109-

L112

```
receive() external payable {
    (bool success, bytes memory data) = getContractAddress("rocke
    if (!success) { revert(getRevertMessage(data)); }
}
```

and forwards it to `minipooldelegate.receiveValidatorBalance`

rocketpool-2.5-Tokenomics-updates/contracts/contract/minipool/RocketMinipoolDelegate.sol:L227-L231

```
require(msg.sender == rocketDAOProtocolSettingsNetworkInterface.g
// Set validator balance withdrawn status
validatorBalanceWithdrawn = true;
// Process validator withdrawal for minipool
rocketNetworkWithdrawal.processWithdrawal{value: msg.value}();
```

Which calculates the `nodeAmount` based on the `ETH` received and submits it as collateral to back the previously minted `nodeAmount` of `nETH`.

rocketpool-2.5-Tokenomics-updates/contracts/contract/network/RocketNetworkWithdrawal.sol:L46-L60

```
uint256 totalShare = rocketMinipoolManager.getMinipoolWithdrawalT
uint256 nodeShare = rocketMinipoolManager.getMinipoolWithdrawalNo
uint256 userShare = totalShare.sub(nodeShare);
// Get withdrawal amounts based on shares
uint256 nodeAmount = 0;
uint256 userAmount = 0;
if (totalShare > 0) {
    nodeAmount = msg.value.mul(nodeShare).div(totalShare);
    userAmount = msg.value.mul(userShare).div(totalShare);
}
// Set withdrawal processed status
rocketMinipoolManager.setMinipoolWithdrawalProcessed(msg.sender);
// Transfer node balance to nETH contract
if (nodeAmount > 0) { rocketTokenNETH.depositRewards{value: nodeA
// Transfer user balance to rETH contract or deposit pool
```

Looking at how the `nodeAmount` of `nETH` that was minted was calculated and comparing it to how `nodeAmount` of `ETH` is calculated, we can observe the following:

- the `nodeAmount` of `nETH` minted is an absolute number of tokens based on the rewards observed by the trusted/oracle nodes. the `nodeAmount` is stored in the storage and later used to calculate the collateral deposit in a later step.
- the `nodeAmount` calculated when depositing the collateral is first assumed to be a `nodeShare` (line 47), while it is actually an absolute number. the `nodeShare` is then turned into a `nodeAmount` relative to the `ETH` supplied to the contract.
- Due to rounding errors, this might not always exactly match the `nETH` minted (see issue 6.34).

- The collateral calculation is based on the `ETH` value provided to the contract. If this value does not exactly match what was reported by the oracle/trusted nodes when minting `nETH`, less/more collateral will be provided.
  - Note: excess collateral will be locked in the `nETH` contract as it is unaccounted for in the `nETH` token contract and therefore cannot be redeemed.
  - Note: providing less collateral will go unnoticed and mess up the 1:1 `nETH:ETH` peg. In the worst case, there will be less `nETH` than `ETH`. Not everybody will be able to redeem their `ETH`.
- Note: keep in mind that the `receive()` function might be subject to gas restrictions depending on the implementation of the withdrawal contract (`.call()` vs. `.transfer()`)

The `nETH` minted is initially uncollateralized and locked to the `minipoolAddress`, which cannot directly redeem it for `ETH`. The next step (next stage) is collateralized with the staking rewards (which, as noted, might not always completely add up to the minted `nETH`). At the last step in `withdraw()`, the `nETH` is transferred to the `withdrawalAddress` of the minipool.

rocketpool-2.5-Tokenomics-updates/contracts/contract/minipool/RocketMinipoolDelegate.sol:L201-L210

```
uint256 nethBalance = rocketTokenNETH.balanceOf(address(this));
if (nethBalance > 0) {
    // Get node withdrawal address
    RocketNodeManagerInterface rocketNodeManager = RocketNodeMana
    address nodeWithdrawalAddress = rocketNodeManager.getNodeWith
    // Transfer
    require(rocketTokenNETH.transfer(nodeWithdrawalAddress, nethB
    // Emit nETH withdrawn event
    emit NethWithdrawn(nodeWithdrawalAddress, nethBalance, block.
}
```

Since the `nETH` initially minted can never take part in the `nETH` token market (as it is locked to the minipool address, which can only transfer it to the withdrawal address in the last step), the question arises why it is actually minted early in the lifecycle of the minipool. At the same time, it could as well be just directly minted to `withdrawalAddress` when providing the right amount of collateral in the last step of the minipool lifecycle. Furthermore, if `nETH` is minted at this stage, it should be questioned why `nETH` is actually needed when you can directly forward the `nodeAmount` to the `withdrawalAddress` instead of minting an intermediary token that is pegged 1:1 to `ETH`.

For reference, `depositRewards` (providing collateral) and `mint` are not connected at all, hence the risk of `nETH` being an undercollateralized token.

rocketpool-2.5-Tokenomics-updates/contracts/contract/token/RocketTokenNETH.sol:L28-L42

```
function depositRewards() override external payable onlyLatestCon
    // Emit ether deposited event
    emit EtherDeposited(msg.sender, msg.value, block.timestamp);
}

// Mint nETH
// Only accepts calls from the RocketMinipoolStatus contract
function mint(uint256 _amount, address _to) override external onl
    // Check amount
    require(_amount > 0, "Invalid token mint amount");
    // Update balance & supply
    _mint(_to, _amount);
    // Emit tokens minted event
    emit TokensMinted(_to, _amount, block.timestamp);
}
```

Recommendation

- It looks like `nETH` might not be needed at all, and it should be discussed if the added complexity of having a potentially out-of-sync `nETH` token contract is necessary and otherwise remove it from the contract system as the `nodeAmount` of `ETH` can directly be paid out to the `withdrawalAddress` in the `receiveValidatorBalance` or `withdraw` transitions.
- If `nETH` cannot be removed, consider minting `nodeAmount` of `nETH` directly to `withdrawalAddress` on `withdraw` instead of first minting uncollateralized tokens. This will also reduce the gas footprint of the Minipool.
- Ensure that the initial `nodeAmount` calculation matches the minted `nETH` and deposited to the contract as collateral (absolute amount vs. fraction).
- Enforce that `nETH` requires collateral to be provided when minting tokens.

6.21 RocketMiniPoolDelegate - on `destroy()` leftover ETH is sent to `RocketVault` where it cannot be recovered Medium ✓ Fixed

Resolution

Leftover ETH is now sent to the node operator address as expected.

https://github.com/ConsenSys/rocketpool-audit-2021-03/blob/0a5f680ae0f4da0c5639a241bd1605512cba6004/rocketpool-rp3.0-updates/contracts/contract/minipool/RocketMinipoolDelegate.sol#L294

Description

When destroying the `MiniPool`, leftover `ETH` is sent to the `RocketVault`. Since `RocketVault` has no means to recover "unaccounted" `ETH` (not deposited via `depositEther`), funds forcefully sent to the vault will end up being locked.

Examples

rocketpool-2.5-Tokenomics-updates/contracts/contract/minipool/RocketMinipoolDelegate.sol:L314-L321

```
// Destroy the minipool
function destroy() private {
    // Destroy minipool
    RocketMinipoolManagerInterface rocketMinipoolManager = Rocket
    rocketMinipoolManager.destroyMinipool();
    // Self destruct & send any remaining ETH to vault
    selfdestruct(payable(getContractAddress("rocketVault")));
}
```

Recommendation

Implement means to recover and reuse `ETH` that was forcefully sent to the contract by `MiniPool` instances.

6.22 RocketDAO - personally identifiable member information (PII) stored on-chain Medium Acknowledged

Resolution

Acknowledged with the following statement:

> This is by design, need them to be publicly accountable. We'll advise their node should not be running on the same machine as their email software though.

Description

Like a DAO user's e-mail address, PII is stored on-chain and can, therefore, be accessed by anyone. This may allow de-pseudonymize users (and correlate Ethereum addresses to user email addresses) and be used for spamming or targeted phishing campaigns putting the DAO users at risk.

Examples

rocketpool-go-2.5-Tokenomics/dao/trustednode/dao.go:L173-L183

```
// Return
return MemberDetails{
    Address: memberAddress,
    Exists: exists,
    ID: id,
    Email: email,
    JoinedBlock: joinedBlock,
    LastProposalBlock: lastProposalBlock,
    RPLBondAmount: rplBondAmount,
    UnbondedValidatorCount: unbondedValidatorCount,
}, nil
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/node/RocketDAONodeTrusted.sol:L110-L112

```
function getMemberEmail(address _nodeAddress) override public vie
    return getString(keccak256(abi.encodePacked(daoNameSpace, "me
}
```

Recommendation

Avoid storing PII on-chain where it is readily available for anyone.

6.23 Rocketpool CLI - Insecure SSH HostKeyCallback Medium ✓ Fixed

Resolution

A proper host key callback function to validate the remote party's authenticity is now defined.

https://github.com/ConsenSys/rocketpool-audit-2021-03/blob/0a5f680ae0f4da0c5639a241bd1605512cba6004/smartnode-1.0.0-rc1/shared/services/rocketpool/client.go#L114-L117

Description

The SSH client factory returns instances that have an insecure `HostKeyCallback` set. This means that SSH servers' public key will not be validated and thus initialize a potentially insecure connection. The function should not be used for production code.

Examples

smartnode-2.5-Tokenomics/shared/services/rocketpool/client.go:L87

```
HostKeyCallback: ssh.InsecureIgnoreHostKey(),
```

6.24 Deployment - Docker containers running as root Medium

Description

By default, Docker containers run commands as the `root` user. This means that there is little to no resistance for an attacker who has managed to break into the container and execute commands. This effectively negates file permissions already set into the system, such as storing wallet-related information with `0600` as an attacker will most likely drop into the container as `root` already.

Examples

Missing `USER` instructions affect both SmartNode Dockerfiles:

smartnode-2.5-Tokenomics/docker/rocketpool-dockerfile:L25-L36

```
# Start from ubuntu image
FROM ubuntu:20.10

# Install OS dependencies
RUN apt-get update && apt-get install -y ca-certificates

# Copy binary
COPY --from=builder /go/bin/rocketpool /go/bin/rocketpool

# Container entry point
ENTRYPOINT ["/go/bin/rocketpool"]
```

smartnode-2.5-Tokenomics/docker/rocketpool-pow-proxy-dockerfile:L24-L35

```
# Start from ubuntu image
FROM ubuntu:20.10

# Install OS dependencies
RUN apt-get update && apt-get install -y ca-certificates

# Copy binary
COPY --from=builder /go/bin/rocketpool-pow-proxy /go/bin/rocketpo

# Container entry point
ENTRYPOINT ["/go/bin/rocketpool-pow-proxy"]
```

Recommendation

In the Dockerfiles, create an unprivileged user and use the `USER` instruction to switch. Only then, the entrypoint launching the SmartNode or the POW Proxy should be defined.

6.25 RocketPoolMinipool - should check for address(0x0) Medium ✓ Fixed

Resolution

Addressed in branch `rp3.0-updates` (rocket-pool/rocketpool@`b424ca1`) by changing requiring that the contract address is not `0x0`.

Description

The two implementations for `getContractAddress()` in `Minipool/Delegate` are not checking whether the requested contract's address was ever set before. If it were never set, the method would return `address(0x0)`, which would silently make all `delegatecall`s succeed without executing any code. In contrast, `RocketBase.getContractAddress()` fails if the requested contract is not known.

It should be noted that this can happen if `rocketMinipoolDelegate` is not set in global storage, or it was cleared afterward, or if `_rocketStorageAddress` points to a contract that implements a non-throwing fallback function (may not even be storage at all).

Examples

- Missing checks

rocketpool-2.5-Tokenomics-updates/contracts/contract/minipool/RocketMinipool.sol:L170-L172

```
function getContractAddress(string memory _contractName) private
    return rocketStorage.getAddress(keccak256(abi.encodePacked("c
}
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/minipool/RocketMinipoolDelegate.sol:L91-L93

```
function getContractAddress(string memory _contractName) private
    return rocketStorage.getAddress(keccak256(abi.encodePacked("c
}
```

- Checks implemented

rocketpool-2.5-Tokenomics-updates/contracts/contract/RocketBase.sol:L84-L92

```
function getContractAddress(string memory _contractName) internal
    // Get the current contract address
    address contractAddress = getAddress(keccak256(abi.encodePack
    // Check it
    require(contractAddress != address(0x0), "Contract not found"
    // Return
    return contractAddress;
}
```

Recommendation

Similar to `RocketBase.getContractAddress()` require that the contract is set.

6.26 RocketDAONodeTrustedAction - ambiguous event emitted in actionChallengeDecide Minor ✓ Fixed

Resolution

Instead of emitting an event even though the challenge period has not passed yet, the function call will now revert if the challenge window has not passed yet.

Description

`actionChallengeDecide` succeeds and emits `challengeSuccess=False` in case the challenged node defeats the challenge. It also emits the same event if another node calls `actionChallengeDecided` before the refute window passed. This ambiguity may make a defeated challenge indistinguishable from a challenge that was attempted to be decided too early (unless the component listening for the event also checks the refute window).

Examples

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/node/RocketDAONodeTrustedActions.sol:L244-L260

```
    // Allow the challenged member to refute the challenge at anyt
    // Is it the node being challenged?
    if(_nodeAddress == msg.sender) {
        // Challenge is defeated, node has responded
        deleteUint(keccak256(abi.encodePacked(daoNameSpace, "memb
    }else{
        // The challenge refute window has passed, the member can
        if(getUint(keccak256(abi.encodePacked(daoNameSpace, "memb
            // Node has been challenged and failed to respond in t
            _memberRemove(_nodeAddress);
            // Challenge was successful
            challengeSuccess = true;
        }
    }
    // Log it
    emit ActionChallengeDecided(_nodeAddress, msg.sender, challen
}
```

Recommendation

Avoid ambiguities when emitting events. Consider throwing an exception in the else branch if the refute window has not passed yet (minimal gas savings; it's clear that the call failed; other components can rely on the event only being emitted if there was a decision.

6.27 RocketDAOProtocolProposals, RocketDAONodeTrustedProposals - unused enum

`ProposalType` Minor ✓ Fixed

Resolution

Addressed in branch `rp3.0-updates` (rocket-pool/rocketpool@`b424ca1`) by removing the unused code.

Description

The enum `ProposalType` is defined but never used.

Examples

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/node/RocketDAONodeTrustedProposals.sol:L29-L35

```
enum ProposalType {
    Invite,             // Invite a registered node to join the tr
    Leave,              // Leave the DAO
    Replace,            // Replace a current trusted node with a n
    Kick,               // Kick a member from the DAO with optiona
    Setting             // Change a DAO setting (Quorum threshold,
}
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/protocol/RocketDAOProtocolProposals.sol:L28-L31

```
enum ProposalType {
    Setting             // Change a DAO setting (Node operator min
}
```

Recommendation

Remove unnecessary code.

6.28 RocketDaoNodeTrusted - Unused events Minor ✓ Fixed

Resolution

Addressed in branch `rp3.0-updates` (rocket-pool/rocketpool@`b424ca1`) by removing the unused code.

Description

The `MemberJoined` `MemberLeave` events are not used within `RocketDaoNodeTrusted`.

Examples

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/node/RocketDAONodeTrusted.sol:L19-L23

```
// Events
event MemberJoined(address indexed _nodeAddress, uint256 _rplBond
event MemberLeave(address indexed _nodeAddress, uint256 _rplBondA
```

Recommendation

Consider removing the events. Note: `RocketDAONodeTrustedAction` is emitting `ActionJoin` and `ActionLeave` event.s

6.29 RocketDAOProposal - expired, and defeated proposals can be canceled Minor ✓ Fixed

Resolution

Proposals can now only be cancelled if they are pending or active.

Description

The `RocketDAOProposal.getState` function defaults a proposal's state to `ProposalState.Defeated`. While this fallback can be considered secure, the remaining code does not perform checks that prevent defeated proposals from changing their state. As such, a user can transition a proposal that is `Expired` or `Defeated` to `Cancelled` by using the `RocketDAOProposal.cancel` function. This can be used to deceive users and potentially bias future votes.

The method emits an event that might trigger other components to perform actions.

Examples

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/RocketDAOProposal.sol:L155-L159

```
} else {
    // Check the votes, was it defeated?
    // if (votesFor <= votesAgainst || votesFor < getVotesRequired(
    return ProposalState.Defeated;
}
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/RocketDAOProposal.sol:L239-L250

```
function cancel(address _member, uint256 _proposalID) override pu
    // Firstly make sure this proposal that hasn't already been exe
    require(getState(_proposalID) != ProposalState.Executed, "Pro
    // Make sure this proposal hasn't already been successful
    require(getState(_proposalID) != ProposalState.Succeeded, "Pr
    // Only allow the proposer to cancel
    require(getProposer(_proposalID) == _member, "Proposal can on
    // Set as cancelled now
    setBool(keccak256(abi.encodePacked(daoProposalNameSpace, "can
    // Log it
    emit ProposalCancelled(_proposalID, _member, block.timestamp)
}
```

Recommendation

Preserve the true outcome. Do not allow to cancel proposals that are already in an end-state like `canceled`, `expired`, `defeated`.

6.30 RocketDAOProposal - preserve the proposals correct state after expiration Minor ✓ Fixed

Resolution

Proposals that have been defeated now will show up as such even when expired. The new default value is `ProposalState.Expired`.

Description

The state of proposals is resolved to give a preference to a proposal being `expired` over the actual result which may be `defeated`. The preference for a proposal's status is checked in order:

```
cancelled? -> executed? -> expired? -> succeeded? -> pending? -> active? ->
defeated (default)
```

Examples

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/RocketDAOProposal.sol:L137-L159

```
if (getCancelled(_proposalID)) {
    // Cancelled by the proposer?
    return ProposalState.Cancelled;
    // Has it been executed?
} else if (getExecuted(_proposalID)) {
    return ProposalState.Executed;
    // Has it expired?
} else if (block.number >= getExpires(_proposalID)) {
    return ProposalState.Expired;
    // Vote was successful, is now awaiting execution
} else if (votesFor >= getVotesRequired(_proposalID)) {
    return ProposalState.Succeeded;
    // Is the proposal pending? Eg. waiting to be voted on
} else if (block.number <= getStart(_proposalID)) {
    return ProposalState.Pending;
    // The proposal is active and can be voted on
} else if (block.number <= getEnd(_proposalID)) {
    return ProposalState.Active;
} else {
    // Check the votes, was it defeated?
    // if (votesFor <= votesAgainst || votesFor < getVotesRequired(
    return ProposalState.Defeated;
}
```

Recommendation

consider checking for `voteAgainst` explicitly and return `defeated` instead of `expired` if a proposal was defeated and is queried after expiration. Preserve the actual proposal result.

6.31 RocketRewardsPool - registerClaimer should check if a node is already disabled before decrementing `rewards.pool.claim.interval.claimers.total.next`

Minor ✓ Fixed

Resolution

In the case a submitted `_claimerAddress` value is invalid or disabled, the function now reverts instead of decrementing `rewards.pool.claim.interval.claimers.total.next`.

Description

The other branch in `registerClaimer` does not check whether the provided `_claimerAddress` is already disabled (or invalid). This might lead to inconsistencies where `rewards.pool.claim.interval.claimers.total.next` is decremented because the caller provided an already deactivated address.

This issue is flagged as `minor` since we have not found an exploitable version of this issue in the current codebase. However, we recommend safeguarding the implementation instead of relying on the caller to provide sane parameters. Registered Nodes

cannot unregister, and Trusted Nodes are unregistered when they leave.

Examples

rocketpool-2.5-Tokenomics-updates/contracts/contract/rewards/RocketRewardsPool.sol:L296-L316

```
function registerClaimer(address _claimerAddress, bool _enabled)
    // The name of the claiming contract
    string memory contractName = getContractName(msg.sender);
    // Record the block they are registering at
    uint256 registeredBlock = 0;
    // How many users are to be included in next interval
    uint256 claimersIntervalTotalUpdate = getClaimingContractUser
    // Ok register
    if(_enabled) {
        // Make sure they are not already registered
        require(getClaimingContractUserRegisteredBlock(contractNa
        // Update block number
        registeredBlock = block.number;
        // Update the total registered claimers for next interval
        setUint(keccak256(abi.encodePacked("rewards.pool.claim.in
    }else{
        setUint(keccak256(abi.encodePacked("rewards.pool.claim.in
    }
    // Save the registered block
    setUint(keccak256(abi.encodePacked("rewards.pool.claim.contra
}
```

Recommendation

Ensure that `getClaimingContractUserRegisteredBlock(contractName, _claimerAddress)` returns `!=0` before decrementing the `.total.next`.

6.32 RocketNetworkPrices - Price feed update lacks block number sanity check Minor ✓ Fixed

Resolution

Addressed in branch `rp3.0-updates` (rocket-pool/rocketpool@`b424ca1`) by only allowing price submissions for blocks in the range of `getPricesBlock() < block < block.number`.

Description

Trusted nodes submit the RPL price feed. The function is called specifying a block number and the corresponding RPL price for that block. If a DAO vote goes through for that block-price combination, it is written to storage. In the unlikely scenario that a vote confirms a very high block number such as `uint(-1)`, all future price updates will fail due to the `require` check below.

This issue becomes less likely the more active members the DAO has. Thus, it's considered a minor issue that mainly affects the

initial bootstrapping process.

Examples

rocketpool-2.5-Tokenomics-updates/contracts/contract/network/RocketNetworkPrices.sol:L53-L54

```
// Check block
require(_block > getPricesBlock(), "Network prices for an equal o
```

Recommendation

The function's `_block` parameter should be checked to prevent large block numbers from being submitted. This check could, e.g., specify that node operators are only allowed to submit price updates for a maximum of x blocks ahead of `block.number`.

6.33 RocketDepositPool - Potential gasDoS in assignDeposits Minor Acknowledged

Resolution
The client acknowledges this issue.

Description

`assignDeposits` seems to be a gas heavy function, with many external calls in general, and few of them are inside the for loop itself. By default, `rocketDAOProtocolSettingsDeposit.getMaximumDepositAssignments()` returns `2`, which is not a security concern. Through a DAO vote, the settings key `deposit.assign.maximum` can be set to a value that exhausts the block gas limit and effectively deactivates the deposit assignment process.

rocketpool-2.5-Tokenomics-updates/contracts/contract/deposit/RocketDepositPool.sol:L115-L116

```
for (uint256 i = 0; i < rocketDAOProtocolSettingsDeposit.getMaxim
    // Get & check next available minipool capacity
```

Recommendation

The `rocketDAOProtocolSettingsDeposit.getMaximumDepositAssignments()` return value could be cached outside the loop. Additionally, a check should be added that prevents unreasonably high values.

6.34 RocketNetworkWithdrawal - ETH dust lockup due to rounding errors Minor ✓ Fixed

Resolution

Addressed in branch `rp3.0-updates` (rocket-pool/rocketpool@`b424ca1`) by calculating `userAmount` as `msg.value - nodeAmount`.

Description

There's a potential `ETH` dust lockup when processing a withdrawal due to rounding errors when performing a division.

Examples

rocketpool-2.5-Tokenomics-updates/contracts/contract/network/RocketNetworkWithdrawal.sol:L46-L55

```
uint256 totalShare = rocketMinipoolManager.getMinipoolWithdrawalT
uint256 nodeShare = rocketMinipoolManager.getMinipoolWithdrawalNo
uint256 userShare = totalShare.sub(nodeShare);
// Get withdrawal amounts based on shares
uint256 nodeAmount = 0;
uint256 userAmount = 0;
if (totalShare > 0) {
    nodeAmount = msg.value.mul(nodeShare).div(totalShare);
    userAmount = msg.value.mul(userShare).div(totalShare);
}
```

Recommendation

Calculate `userAmount` as `msg.value - nodeAmount` instead. This should also save some gas.

6.35 RocketAuctionManager - calcBase should be declared constant Minor ✓ Fixed

Resolution

Addressed in branch `rp3.0-updates` (rocket-pool/rocketpool@`b424ca1`) by declaring `calcBase` as `constant`. We would like to note that the variable is re-declared multiple times.

Description

Declaring the same constant value `calcBase` multiple times as local variables to some methods in `RocketAuctionManager` carries the risk that if that value is ever updated, one of the value assignments might be missed. It is therefore highly recommended to reduce duplicate code and declare the value as a public constant. This way, it is clear that the same `calcBase` is used throughout the contract, and there is a single point of change in case it ever needs to be changed.

Examples

rocketpool-2.5-Tokenomics-updates/contracts/contract/auction/RocketAuctionManager.sol: L136-L139

```
function getLotPriceByTotalBids(uint256 _index) override public v
    uint256 calcBase = 1 ether;
    return calcBase.mul(getLotTotalBidAmount(_index)).div(getLotT
}
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/auction/RocketAuctionManager.sol: L151-L154

```
function getLotClaimedRPLAmount(uint256 _index) override public v
    uint256 calcBase = 1 ether;
    return calcBase.mul(getLotTotalBidAmount(_index)).div(getLotC
}
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/auction/RocketAuctionManager.sol: L173-L174

```
// Calculation base value
uint256 calcBase = 1 ether;
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/auction/RocketAuctionManager.sol: L216-L217

```
uint256 bidAmount = msg.value;
uint256 calcBase = 1 ether;
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/auction/RocketAuctionManager.sol: L247-L249

```
// Calculate RPL claim amount
uint256 calcBase = 1 ether;
uint256 rplAmount = calcBase.mul(bidAmount).div(currentPrice);
```

Recommendation

Consider declaring `calcBase` as a private const state var instead of re-declaring it with the same value in multiple, multiple functions. Constant, literal state vars are replaced in a preprocessing step and do not require significant additional gas when accessed than normal state vars.

6.36 RocketDAO* - daoNamespace is missing a trailing dot; should be declared constant/immutable Minor ✓ Fixed

Resolution

Addressed in branch `rp3.0-updates` (rocket-pool/rocketpool@`b424ca1`) by adding a trailing dot to the `daoNamespace`.

Description

`string private daoNameSpace = 'dao.trustednodes'` is missing a trailing dot, or else there's no separator when concatenating the namespace with the vars.

Examples

- requests `dao.trustednodesmember.index` instead of `dao.trustednodes.member.index`

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/node/RocketDAONodeTrusted.sol:L83-L86

```
function getMemberAt(uint256 _index) override public view returns
    AddressSetStorageInterface addressSetStorage = AddressSetStor
    return addressSetStorage.getItem(keccak256(abi.encodePacked(d
}
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/node/RocketDAONodeTrustedActions.sol:L32-L33

```
// The namespace for any data stored in the trusted node DAO (do nc
string private daoNameSpace = 'dao.trustednodes';
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/node/RocketDAONodeTrustedProposals.sol:L22-L26

```
// Calculate using this as the base
uint256 private calcBase = 1 ether;

// The namespace for any data stored in the trusted node DAO (do nc
string private daoNameSpace = 'dao.trustednodes';
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/dao/protocol/RocketDAOProtocol.sol:L12-L13

```
// The namespace for any data stored in the network DAO (do not cha
string private daoNameSpace = 'dao.protocol';
```

Recommendation

Remove the `daoNameSpace` and add the prefix to the respective variables directly.

6.37 RocketVault - consider rejecting zero amount deposit/withdrawal requests Minor ✓ Fixed

Resolution

Addressed in branch `rp3.0-updates` (rocket-pool/rocketpool@`b424ca1`) by requiring that `ETH` and `tokenAmounts` are not zero.

Note that components that used to raise no exception when attempting to deposit/withdraw/transfer zero amount tokens/ETH may now throw which can be used to block certain functionalities (slashAmount==0).

The client provided the following statement:

> We'll double check this. Currently the only way a slashAmount is 0 is if we allow node operators to not stake RPL (min 10% required currently). Though there isn't a check for 0 in the slash function atm, I'll add one now just as a safety check.

Description

Consider disallowing zero amount token transfers unless the system requires this to work. In most cases, zero amount token transfers will emit an event (that potentially triggers off-chain components). In some cases, they allow the caller without holding any balance to call back to themselves (pot. reentrancy) or the caller provided token address.

- `depositEther` allows to deposit zero ETH
  - emits `EtherDeposited`
- `withdrawEther` allows to withdraw zero ETH
  - calls back to `withdrawer` (`msg.sender`)!
  - emits `EtherWithdrawn`
- (`depositToken` checks for amount `>0`)
- `withdrawToken` allows zero amount token withdrawals
  - calls into user provided (actually a network contract) `tokenAddress`)
  - emits `TokenWithdrawn`
- `transferToken` allows zero amount token transfers
  - emits `TokenTransfer`

Examples

rocketpool-2.5-Tokenomics-updates/contracts/contract/RocketVault.sol:L50-L57

```
function depositEther() override external payable onlyLatestNetwo
    // Get contract key
    bytes32 contractKey = keccak256(abi.encodePacked(getContractN
    // Update contract balance
    etherBalances[contractKey] = etherBalances[contractKey].add(m
    // Emit ether deposited event
    emit EtherDeposited(contractKey, msg.value, block.timestamp);
}
```

Recommendation

Zero amount transfers are no-operation calls in most cases and should be avoided. However, as all vault actions are authenticated (to registered system contracts), the risk of something going wrong is rather low. Nevertheless, it is recommended to deny zero amount transfers to avoid running code unnecessarily (gas consumption), emitting unnecessary events, or potentially call back to callers/token address for ineffective transfers.

6.38 RocketVault - methods returning static return values and unchecked return parameters Minor

✓ Fixed

Resolution

The unused boolean return values have been removed and reverts have been introduced instead.

Description

The `Token*` methods in `RocketVault` either throw or return `true`, but they can never return `false`. If the method fails, it will always throw. Therefore, it is questionable if the static return value is needed at all. Furthermore, callees are in most cases not checking the return value of

Examples

- static return value `true`

rocketpool-2.5-Tokenomics-updates/contracts/contract/RocketVault.sol:L93-L96

```
// Emit token transfer
emit TokenDeposited(contractKey, _tokenAddress, _amount, block.ti
// Done
return true;
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/RocketVault.sol:L113-L115

```
emit TokenWithdrawn(contractKey, _tokenAddress, _amount, block.ti
// Done
return true;
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/RocketVault.sol:L134-L137

```
// Emit token withdrawn event
emit TokenTransfer(contractKeyFrom, contractKeyTo, _tokenAddress,
// Done
return true;
```

- return value not checked

rocketpool-2.5-Tokenomics-updates/contracts/contract/node/RocketNodeStaking.sol:L149-L150

```
rocketVault.depositToken("rocketNodeStaking", rplTokenAddress, _a
// Update RPL stake amounts & node RPL staked block
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/auction/RocketAuctionManager.sol:L252-L252

```
rocketVault.withdrawToken(msg.sender, getContractAddress("rocketT
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/node/RocketNodeStaking.sol:L172-L172

```
rocketVault.withdrawToken(msg.sender, getContractAddress("rocketT
```

rocketpool-2.5-Tokenomics-updates/contracts/contract/node/RocketNodeStaking.sol:L193-L193

```
rocketVault.transferToken("rocketAuctionManager", getContractAddr
```

Recommendation

Define a clear interface for these functions. Remove the static return value in favor of having the method throw on failure (which is already the current behavior).

6.39 Deployment - Overloaded Ubuntu base image

Minor

Description

The SmartNode and the corresponding proxy Dockerfiles base their builds on the `ubuntu:20.10` image. This image introduces many unrelated tools that significantly increase the container's attack surface and the tools an attacker has at their disposal once they have gained access to the container. Some of these tools include:

- apt
- bash/sh
- perl

Examples

smartnode-2.5-Tokenomics/docker/rocketpool-dockerfile:L26-L26

```
FROM ubuntu:20.10
```

smartnode-2.5-Tokenomics/docker/rocketpool-pow-proxy-dockerfile:L25-L25

```
FROM ubuntu:20.10
```

Recommendation

Consider using a smaller and more restrictive base image such as Alpine. Additionally, AppArmor or Seccomp policies should be used to prevent unexpected and potentially malicious activities during the container's lifecycle. As an illustrative example, a SmartNode container does not need to load/unload kernel modules or loading a BPF to capture network traffic.

6.40 RocketMinipoolDelegate - enforce that the delegate contract cannot be called directly Minor

✓ Fixed

Resolution

Addressed in branch `rp3.0-updates` (rocket-pool/rocketpool@`b424ca1`) by removing the constructor and therefore the initialization code from the RocketMinipoolDelegate contract. The contract cannot be used directly anymore as all relevant methods are decorated `onlyInitialised` and there is no way to initialize it in the implementation directly.

Description

This contract is not meant to be consumed directly and will only be delegate called from `Minipool`. Being able to call it directly might even create the problem that, in the worst case, someone might be able to `selfdestruct` the contract rendering all other contracts that link to it dysfunctional. This might even not be easily detectable because `delegatecall` to an EOA will act as a NOP.

The access control checks on the methods currently prevent methods from being called directly on the delegate. They require state variables to be set correctly, or the delegate is registered as a valid minipool in the system. Both conditions are improbable to be

fulfilled, hence, mitigation any security risk. However, it looks like this is more of a side-effect than a design decision, and we would recommend not explicitly stating that the delegate contract cannot be used directly.

Examples

rocketpool-2.5-Tokenomics-updates/contracts/contract/minipool/RocketMinipoolDelegate.sol:L65-L70

```
constructor(address _rocketStorageAddress) {
    // Initialise RocketStorage
    require(_rocketStorageAddress != address(0x0), "Invalid stora
    rocketStorage = RocketStorageInterface(_rocketStorageAddress)
}
```

Recommendation

Remove the initialization from the constructor in the delegate contract. Consider adding a flag that indicates that the delegate contract is initialized and only set in the Minipool contract and not in the logic contract (delegate). On calls, check that the contract is initialized.

Appendix 1 - Disclosure

ConsenSys Diligence (“CD”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via ConsenSys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any Third-Party by virtue of publishing these Reports.

PURPOSE OF REPORTS The Reports and the analysis described therein are created solely for Clients and published with their

consent. The scope of our review is limited to a review of Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond Solidity that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) – on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

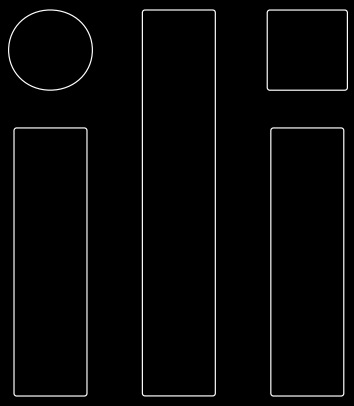
LINKS TO OTHER WEB SITES FROM THIS WEB SITE You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

TIMELINESS OF CONTENT The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.

Request a Security Review Today

Get in touch with our team to request a quote for a smart contract audit.

CONTACT US



Subscribe to Our Newsletter

Stay up-to-date on our latest offerings, tools, and the world of blockchain security.

Email*

e-mail address

→