Service staking (part of AIP-1)

22th September 2023

Mariapia Moscatiello, David Minarsch,
Andrey Lebedev, Aleksandr Kuperman

TL;DR

In our pursuit of implementing the triple lock (AIP-1), we introduce a staking module allowing anyone, including OLAS DAO, to deploy staking contracts for autonomous services. This empowers service owners to incentivize agent operators using any ERC-20 token or ETH.

Introduction

In our pursuit of implementing the triple lock outlined in AIP-1, there is a critical missing element not yet implemented into the current protocol: a reward mechanism for stakers. In this document, we introduce a straightforward staking module that empowers individuals, including OLAS DAO, to deploy staking contracts for autonomous services. This module facilitates in turn service owners, including OLAS DAO, to incentivize service agent operators using any ERC-20 token or ETH.

Overview of the proposed design

The proposed simple setup allows anyone to deploy a staking contract for services, that, in turn would allow service owners to incentivise autonomous service operation with any ERC20 token or ETH.

Contract Features.

- Each service staking contract is only configurable at the time of deployment, making it immutable once deployed. No administrative access is available. Configuration updates necessitate the deployment of new staking contracts.
- Each service staking contract provides staking rewards in the token selected by the contract deployer only.
- Staking contracts should directly receive their staking rewards from whoever funds it (e.g. Olas DAO or anyone else). Staking contracts may need periodic infusions of funds to accommodate growing demand.
- There's a maximum amount of stakers the contract permits at any given time, this is specified during contract deployment.

Service Requirements.

- Service owners can "stake" their service, effectively transferring their service into the staking smart contract. The service must meet the following criteria to participate.
    - Transfer approval: The Service Owner must approve the staking contract before being able to stake their service NFT.
    - Agent Instances: the service should be configured with the correct number of agent instances as specified on the staking contract.
    - Staked Amount and Security Deposit: The Service Owner and Operator must have deposited their security deposit and their *security bond* into either
        - ServiceRegistry (if the service is secured with ETH)
        - ServiceRegistryToken (if the service is secured with an ERC20 token)

      Moreover, the security deposit has to be larger or equal that the *minimum stake* required, where the *minimum stake* is defined by the staking contract deployer.

      Note that, when service is secured with ETH (resp. ERC20), the service owner can only participate in the ETH (resp. ERC20) staking rewards.
    - While a service is staked the service accrues a yield (e.g. 5% ar any other percentage set by the staking contract deployer) in the token corresponding to the service-owner security deposit one (e.g. OLAS, ETH, or any other ERC-20). The yield is configurable during contract initialization and remains fixed. Once the staking contract funds are used up, no further

yield is earned. Yield is only earned by services with a minimum number of transactions per epoch (a fixed number of blocks determined by the staking contract deployer). This epoch begins counting from the service staking point. If the service is staked for less than an epoch, no yield is earned for that epoch.Transaction count is determined by checking the nonce on the service's multisig. Further details on reward criteria are provided below.

- Once a service is unstaked, an operation that can be executed at any time, any accrued staking rewards are transferred to the service's multisig.

Happy path for staking mechanism

In this section, we outline the ideal scenario for staking mechanism. We also highlight key considerations and steps to ensure the smooth operation of the staking mechanism.

1. **Deploy the staking contract:** The contract should be immutable and not-ownable to ensure trust and transparency in the staking process.
2. **Fund the staking contract:** Staking contracts should directly receive their staking rewards from whoever funds them (e.g. Olas DAO or anyone else). Staking contracts may need periodic infusions of funds to accommodate growing demand. The funded amount and the staking rewards determine the maximum number of services that can participate in the staking.
3. ***Stake(serviceId):*** This contract has a *Stake(serviceId)* method that can be called by Service Owner when their service is deployed. This method calls *safeTransferFrom(service owner address, address_staking_contract, serviceID)* after receiving approval from the service owner. This method requires the Service Owner to ensure several things before proceeding:
    a. **Approval:** The Service Owner must approve the staking contract before calling the Stake method.
    b. **Agent Instances:** The Service Owner should ensure they have the correct number of agent instances (1 initially) as specified on the staking contract.
    c. **Staked Amount and Security Deposit:** The Service Owner and Operator must have deposited their *security deposit* and their *security bond* into either
        i. ServiceRegistry (if the service is secured with ETH)
        ii. ServiceRegistryToken (if the service is secured with an ERC20 token)

    Moreover, the security deposit has to be larger or equal that the *minimum stake* required, where the *minimum stake* is defined by the staking contract deployer.

    Note that, when service is secure with ETH (resp. ERC20), the service owner can only participate in the ETH (resp. ERC20) staking rewards.

The staking contract needs to record the block timestamp and the current nonce of the service's multisig for future reward calculation.

   d. **Deployed State**: The service must be in the deployed state.

4. ***Unstake(serviceId)***. This contract has an *Unstake(serviceId)* method that the user can call at any time. When called, the staking module checks for criteria that must be met before sending incentives to the service multisig. To this end, the contract uses the stored block timestamp and the current block timestamp, as well as the stored nonce and the current nonce of the multisig, to compute the number of blocks for which the service was staked. One possible reward criterion is discussed in a note below.

Multiple operator staking module

While the on-chain part of the staking module remains similar for multiple agent instances, additional off-chain work is needed to ensure fair distribution of staking rewards among operators. Specifically, the agent instances control the funds into the service multisig, hence, just by sending the eventual staking rewards to the service multisig, would allow the agent instances to control their staking rewards.

Security deposit amounts

With the current ServiceRegistry, ServiceRegistryToken, and ServiceManagerToken implementation, the service owner can only stake the total *security deposit* amount. The staking contract checks that the security deposit is larger or equal than the *minimum stake* required, where the *minimum stake* is defined by the staking contract deployer.

Note on reward criteria

As mentioned above, once service owner calls redeem, the staking module checks whether

$(nonce(redeem\ time) - nonce(staking\_time))/staking\_time \geq x,$

for a certain fixed x, where x is defined by the staking contract deployer.

- If the criterion is satisfied, the staking rewards are sent to the corresponding service multisig and the service owner will receive back the staked service token with serviceID.
- If the criterion is not satisfied, the service owner will receive back the staked service token with serviceID and no staking reward is sent.