Contracts vulnerabilities

Vulnerabilities list: #1

Involved contracts and level of the bugs

The present document describes issues affecting Tokenomics contracts

Vulnerabilities

1. `depositServiceDonationsETH` function (services state)

Severity: Low

The following function is implemented in the Treasury contract:

```
function depositServiceDonationsETH(uint256[] memory serviceIds, uint256[]
memory amounts) external payable
```

This service donating function calls another function from the Tokenomics contract that ultimately results in calling the internal function `_trackServiceDonations()`. The latter one checks whether agent and component Ids of each of the passed service Id exist, and if not, reverts with the `ServiceNeverDeployed()` error. The error arises from the fact that the service was never deployed, and its underlying component and agent Ids were not assigned (the assignment of underlying component and/or agent Ids to a service happens during the deployment of the service itself).

However, after a specific service is deployed at least once and then terminated, it can be updated and re-deployed again. In particular, the service can be updated with a different

set of agent Ids, making the donation distribution setup invalid for the following reason. If this updated service receives a donation before it is re-deployed, the donation will be distributed between its old component and agent Ids owners and not the new ones.

Therefore, donating to an updated service before its redeployment can affect the correct distribution of rewards in the Tokenomics contract. We recommend not to donate when a service is not in the `Deployed` or `TerminatedBonded` state (e.g. any service with *serviceIds[i]* not in `Deployed` or `TerminatedBonded` state must not be passed as input parameters to the function **depositServiceDonationsETH**). The state of the service can be easily checked via the ServiceRegistry contract view function `getService(uint256 serviceId)`.

2. `depositServiceDonationsETH` function (OLAS incentives)

Severity: Informative

The following function is implemented in the Treasury contract:

```
function depositServiceDonationsETH(uint256[] memory serviceIds, uint256[]
memory amounts) external payable
```

When a DAO member holding the veOLAS threshold[1] uses this method to donate ETH to a certain service, the owners of the agents and components referenced in such a service are entitled to receive a share of the donation and OLAS tokens arising from inflation.

While the current approach encourages service registration and donations through the utilization of all available OLAS each epoch, this might be utilized in a counter-intended way by malicious donators. If a donator owns all the underlying components and agents, meets the sufficient veOLAS requirement, and makes only a small donation to their service, they could accrue a significant number of OLAS tokens through inflation top-ups at a low cost. This behavior may yield considerable gains initially but becomes less profitable as more major players utilize the protocol, leading to more donations being distributed among multiple services and stakeholders.

[1] Currently, the threshold for participation is set at 10000 veOLAS, and adjustments to this threshold can be made through a governance voting process.

3. `Checkpoint` function

Severity: Informative

The following function is implemented in the Tokenomics contract:

```
function checkpoint() external returns (bool)
```

The purpose of this function is to record the global data and update tokenomics parameters and/or fractions when `changeTokenomicsParameters()` and/or `changeIncentiveFractions()` methods are called.

When the epoch following the settled epoch has a year change, the function performs an incorrect calculation of top-ups for the event emit. Specifically, the emitted top-ups value is overwritten with the one calculated for the next epoch. This issue is considered informative because the amount of top-ups to be allocated (and further minted) is calculated correctly; the problem lies only with the emitted amount.
By addressing this issue, the Tokenomics contract will provide accurate information regarding the allocation of top-ups.

4. `getLastIDF` function

Severity: Informative

The following function is implemented in the Tokenomics contract and used in GenericBondCalculator contract:

```
function getLastIDF() external view returns (uint256 idf)
```

This function retrieves the inverse discount factor (IDF) from the epoch just prior to the latest checkpoint, expressed as a multiple of 1e18. The calculation of IDF pertains to the current epoch and draws from the outcomes of the previous epoch. It's worth noting that if the function were modified to output getIDF(epochCounter) instead of getIDF(epochCounter-1), the getLastIDF() function would have more prominently represented the performance results from the most recent epoch.
In absence of redeploying a new contract, we recall that getLastIDF gives more prominently information associated with performance in the second to latest settled epoch, hence we suggest using getIDF(epochCounter) to check the performance prominently represented the results from the most recent settled epoch.