Contracts vulnerabilities

Vulnerabilities list

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
```

If a DAO member, holding the veOLAS threshold[1], uses this method to donate ETH to a specific service, or if the service owner is a DAO member holding the veOLAS threshold[2], the owners of the agents and components referenced in that service are entitled to receive a share of the donation and OLAS tokens generated through 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 or malicious service-owners. If a donator (or the service-owner) owns all the underlying components and agents, meets the sufficient veOLAS requirement, and makes only a small donation to their service, they could accrue

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

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.

3. `Checkpoint` function - event

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.

5. `epochLen`

Severity: Low

With the current tokenomics implementation, the method `changeTokenomicsParameters()` enables the selection of the epochLen parameter, allowing any value between `MIN_EPOCH_LENGTH` and one year.

However, if epochLen is set precisely to one year, a potential problem arises in the `checkpoint()` method. This issue comes from the fact that the checkpoint can only succeed if called after the expiration of epochLen from the previous checkpoint, but not later than one year. Given the discrete nature of block time, achieving a one-year time difference from the previous checkpoint call is highly improbable.

To address this concern without the need for contract redeployment, it is recommended to avoid setting epochLen to one year. Specifically, it is suggested to choose a value slightly below one year, such as one year minus one day in block time. This adjustment ensures the successful execution of the `checkpoint()` method within the constraints of block time, mitigating the potential issue described above.

6. `deposit` method

Severity: High

In the depository contracts, the following method is implemented:

```
function deposit(uint256 productId, uint256 tokenAmount) external
```

This method allows users to deposit tokens, acquiring OLAS tokens at a discounted rate. A potential concern can arise ten years after OLAS token launch in the case of an epoch crossing into year intervals. In this scenario, a portion of OLAS becomes mintable only in the eleventh year, as a result of the 1 billion fixed supply constraint for the initial ten years.

The creation of bonding programs with payouts leading to exceeding the total OLAS supply mintable before ten years and the bonder's depositing the full amount expecting

these payouts lead to a silent return in the OLAS `mint()` method and not a revert. This results in successful product deposit and a consequent loss of OLAS payouts for bonders.

To address this, a more specific check for epoch crossing year intervals can be integrated into the tokenomics `checkpoint()` method. In the absence of redeploying a new contract, it is recommended to carefully propose the creation of bonding programs at the end of the tenth year. These programs should be structured ensuring that the payouts are designed to keep the total amount of OLAS minted below 1 billion OLAS before the ten-year mark. This precautionary measure prevents eventual lost OLAS payouts.

7. `checkpoint` method - cross-year

Severity: Informative

In the tokenomics contracts, the following method is implemented:

```
function checkpoint() external
```

This method allows users to deposit tokens, acquiring OLAS tokens at a discounted rate. A potential concern may arise in the event of an epoch crossing into year intervals, where a portion of OLAS larger than the year inflation limit becomes mintable.

The creation of bonding programs with payouts leading to an excess of the total OLAS mintable before the specified year and the bonder depositing the full amount may result in an amount of minted OLAS exceeding the year inflation limit. It's crucial to note that, at most, only the amount reserved for the remaining time of the epoch from the following year can be minted.

To address this, a more specific check for epoch crossing year intervals can be integrated into the tokenomics `checkpoint()` method. In the absence of redeploying a new contract, it is recommended to carefully propose the creation of bonding programs for epoch-crossing years. These programs should be structured to ensure that the payouts are designed in a manner that keeps the total amount of OLAS minted below the year inflation limit.

8. Treasury Fund Token Management

Severity: Informative

By design, within the Treasury contract, there is currently no mechanism in place to facilitate the removal of tokens other than ETH that have not been added to the Treasury through the treasury *depositTokenForOLAS()* method.

Therefore, we strongly recommend refraining from transferring funds directly to the Treasury contract that does not adhere to the established tokenomics logic. This precautionary measure will help prevent potential freezing of funds within the Treasury contract