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 top-ups 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

[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.

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 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. `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.

4. `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.

5. 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

6. Encoded inflation schedule

Severity: Informative

If donors in a given epoch fail to meet the veOLAS threshold for donating ETH to specific services within 10 years of OLAS token creation, the reserved OLAS inflation for top-ups

remains inactive. Although accounted for in the inflation schedule of that epoch, that amount is essentially deducted from the inflation schedule. For instance, if x OLAS were accounted for in the inflation for top-ups during the inaugural tokenomics epoch but no donator meets the veOLAS threshold, these top-ups cannot be utilized for subsequent epochs encoded in the 10-year inflation schedule.

A similar scenario can occur when OLAS top-ups and staking incentives are distributed. Due to the natural rounding behavior of Solidity and the division involved in calculating top-ups and staking emissions, it's possible that the actual sum of OLAS allocated to owners of agents and components referenced in donated services and the calculated staking emissions might be slightly less than the exact amount that can be extracted from the encoded inflation schedule in the tokenomics contract. In such cases, the difference between the exact amount and the actually allocated amount for top-up and staking is implicitly deducted from the inflation schedule.

This deferred inflation isn't lost; rather, it's postponed, as the OLAS token ensures that no more than 1 billion tokens are minted within a decade, with no more than 2% of the supply cap being minted annually, starting from 1 billion.

7. Withheld tokens

Severity: Informative

The TargetStakingDispenser contract on L2 withholds some staking emissions sent by L1 (see the section “Verification on staking contract enabled by StakingVerifier” here for details on the tokens withheld by the TargetStakingDispenser).

To prevent L1 from sending new emissions while there are still withheld emissions on the TargetStakingDispenser, we need to ensure regular synchronizations between L1 and L2. Specifically, if there is demand for emissions for a specific contract on L2, and L1 is synchronized with the withheld amount on the TargetStakingDispenser, L1 will only send a message without minting or sending new emissions to the L2 target contract until the withheld amount is fully utilized and additional demand arises.

Additionally, if there is no new demand for emissions from the L2 target dispenser and a withheld amount remains, the DAO can initiate a new staking campaign to utilize the withheld amount.

Finally, the DAO can employ the combination of the functions **migrate()**, **syncWithheldAmount()**, **processDataMaintenance()**,

updateWithheldAmountMaintenance() to transfer and update balance of the withheld tokens to a DAO-controlled account.

8. `changeManagers` function (specifically - voteWeighting)

Severity: Informative

The following function is implemented in the Dispenser contract:

```
   function changeManagers(address _tokenomics, address _treasury,
address _voteWeighting) external
```

The purpose of this function is to change core tokenomics contract addresses. However, when the Vote Weighting contract address is changed, if not all the staking incentives are claimed, those can be lost. The idea is to force claim all the staking incentives before the voteWeighting is updated. More details here.

9. `claimStakingIncentives` / `_calculateStakingIncentivesBatch` functions

Severity: Low

Following functions is implemented in the Dispenser contract:

```
    function claimStakingIncentives( uint256 numClaimedEpochs,
uint256 chainId, bytes32 stakingTarget, bytes memory bridgePayload )
external payable
```

```
    function _calculateStakingIncentivesBatch(uint256
numClaimedEpochs, uint256[] memory chainIds, bytes32[][] memory
stakingTargets ) internal returns (uint256[] memory totalAmounts,
uint256[][] memory stakingIncentives, uint256[] memory
transferAmounts)
```

The purpose of these functions is to calculate staking incentives and returns according to the staking target provided. However, these functions do not account for the fact that the amount of OLAS previously sent to L2 and communicated as not used and available for re-usage (`withheldAmount`) should also be subtracted from the staking incentives amount in favor of amounts returned back to Tokenomics. This means that staking

incentives amounts that are reused from withheld ones are calculated as subject to inflation used, whereas in fact that part of inflation is untouched. Ultimately it results in spending less inflation throughout the inflation period for the amount of funds that were minted but withheld on L2 target dispenser contracts as over-excessive.

Note that the inflation amount is not returned to Tokenomics due to `withheldAmount` reuse is never minted, meaning there is no loss of funds, just the inflation miscalculation lowering its yearly mint possibility. In the absence of redeploying a new contract, the DAO might act to adjust the inflation numbers in a distant timeline consolidating information about all the withheld amounts across chains.

10. `migrate` function

Severity: Low

The following function is implemented in the TargetDispenserL2 contract:

```
function migrate(address newL2TargetDispenser) external
```

The purpose of this function is to migrate all the funds to a new L2TargetDispenser address. However, this function does not check if the current `withheldAmount` value is zero before migrating, essentially having the possibility to lose the inflation information for not sending additional funds to L2.

In order to avoid the loss of `withheldAmount`, the DAO is advised to update the value with the **updateWithheldAmountMaintenance()** function call right after the TargetDispenser migration procedure is complete.

11. `_sendMessage` function

Severity: Low

The following function is implemented in the OptimismDepositProcessorL1 contract:

```
function _sendMessage(address[] memory targets, uint256[] memory
stakingIncentives, bytes memory bridgePayload, uint256
transferAmount, bytes32 batchHash) internal override returns (uint256
sequence, uint256 leftovers)
```

This function forms required data to send tokens and messages to L2 in all the optimism deposit processor related contracts. A user-controlled gas limit is decoded as a uint256, which is later truncated to uint32 when passed to the CrossDomainMessenger. If a user supplies a payload with a value exceeding type(uint32).max, the truncation produces a much smaller gas limit than intended, bypassing the protocol's minimum gas check.

Although this action does not result in loss of funds (which are sent separately), it could deliberately pass a smaller amount of gas such that a corresponding function on L2 reverts. This can then be corrected via the **processDataMaintenance()** function. In the absence of contract re-deployment, users are advised to pass a sufficient amount of gas, or just have it set to zero, such that the fallback value takes care of it.