Contracts vulnerabilities

Vulnerabilities list

Contracts vulnerabilities **1**
Vulnerabilities list 1
Involved contracts and level of the bugs 1
Vulnerabilities 1
1. depositServiceDonationsETH function (services state) 1
2. depositServiceDonationsETH function (OLAS incentives) 2
3. deposit method 3
4. checkpoint method - cross-year 3
5. Treasury Fund Token Management 4
6. Encoded inflation schedule 4
7. Withheld tokens 5
8. Checkpoint function - event (only affecting old contract version) 6
9. getLastIDF function (only affecting old contract version) 6
10. epochLen (only affecting old contract version) 7
11. changeManagers function (specifically - voteWeighting) 7
12. calculateStakingIncentives function 8
13. retain function 8
14. onTokenBridged function 8
15. _sendMessage function (OptimismDepositProcessorL1 and OptimismTargetDispenserL2) 9
16. syncWithheldAmountMaintenance function 9
17. _sendMessage function (ArbitrumTargetDispenserL2 and others) 10
18. _sendMessage function (WormholeDepositProcessorL1) 10
19. _sendMessage function (ArbitrumDepositProcessorL1) 10
20. _sendMessage function (ArbitrumDepositProcessorL1 and WormholeDepositProcessorL1) 11
21. retain function 11

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 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.

[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.

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()**, **syncWithheldTokens()**, and **processDataMaintenance()** to transfer the withheld tokens to a DAO-controlled account.

8. `Checkpoint` function - event `(only affecting` [`old contract version`]`)`

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.

9. `getLastIDF` function `(only affecting` [`old contract version`]`)`

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.

10. `epochLen (only affecting` [`old contract version`]`)`

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.

11. `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.

12. `calculateStakingIncentives` function

Severity: High

The following function is implemented in the Dispenser contract:

```
   function calculateStakingIncentives(uint256 numClaimedEpochs,
uint256 chainId, bytes32 stakingTarget, uint256 bridgingDecimals)
public returns (uint256 totalStakingIncentive, uint256
totalReturnAmount, uint256 lastClaimedEpoch, bytes32 nomineeHash)
```

The purpose of this function is to calculate staking incentives for the number of claimed epochs. However, there could be a situation when all the staking weights are zero along with the overall total weight sum, which is not accounted for in the code. The solution is to insert a logic accounting for epochs not available for staking incentives such that all the inflation is returned back to the tokenomics. More details here.

13. `retain` function

Severity: High

The following function is implemented in the Dispenser contract:

```
   function retain() external
```

The purpose of this function is to retain staking inflation by voting for a specific retainer address, such that the retained inflation is returned back to the tokenomics. However, the function implementation skips the retainer nominee checkpoint action, which results in incorrect retainer amount calculation. More details here.

14. `onTokenBridged` function

Severity: High

The following function is implemented in the GnosisTargerDispenserL2 contract:

```
   function onTokenBridged(address, uint256, bytes calldata data)
external
```

The purpose of this function is to execute the provided payload after the token is delivered cross-chain using the OmniBridge. However, this callback does not provide the originating L1 sender address, thus not being able to pass the validation on the L2 side. This means that anyone is able to manipulate the data when sending tokens across the bridge. The solution is to transfer tokens separately from the message transfer originating on L1 and drop the usage of this function on L2. More details here.

15. `_sendMessage` function (OptimismDepositProcessorL1 and OptimismTargetDispenserL2)

Severity: Medium

The following function is implemented in OptimismDepositProcessorL1 and OptimismTargerDispenserL2 contracts:

```
   function _sendMessage(...)
```

The purpose of this function is to send messages from L1 to L2 and back. However, the cost parameter passed as part of a payload is unnecessary in Optimism cross-chain interactions. Hence, the msg.value checked for the provided cost value is not needed, meaning that the additional msg.value should not be passed along with the function call. The gas limit can also be problematic when set by the sender. The solution is to remove the cost variable completely and not provide cost as a value parameter when calling the Optimism relayer contract. Plus, make gas limit minimum caps variables much higher as they have been computed to be sufficient for message deliveries on both L1 and L2. More details here.

16. `syncWithheldAmountMaintenance` function

Severity: Medium

The following function is implemented in the Dispenser contract:

```
   function syncWithheldAmountMaintenance(uint256 chainId, uint256
amount) external
```

The purpose of this function is to sync withheld amounts that failed to be delivered from L2 to L1. However, this is prone to a replay execution if the maintenance sync took place, but the bridge ultimately recovered the data and delivered the message that was considered irrecoverable. The solution is to use nonces and compute unique hashes with corresponding L2 contract addresses in order for the unexpected recovered messages not to be replayed. More details here.

17. `_sendMessage` function (ArbitrumTargetDispenserL2 and others)

Severity: Medium

The following function is implemented in the ArbitrumTargerDispenserL2 contract:

```
   function _sendMessage(...)
```

The purpose of this function is to send messages from L2 to L1. However, there is no check that the msg.value passed is not bigger than zero. All the msg.value leftovers must be returned back to the sender. More details here and here.

18. `_sendMessage` function (WormholeDepositProcessorL1)

Severity: Medium

The following function is implemented in the WormholeDepositProcessorL1 contract:

```
   function _sendMessage(...)
```

The purpose of this function is to send messages from L1 to L2. However, the refund chain Id is not provided in the wormhole chain Id classification. The solution is to change it to the wormhole one. More details here.

19. `_sendMessage` function (ArbitrumDepositProcessorL1)

Severity: Medium

The following function is implemented in the ArbitrumDepositProcessorL1 contract:

```
function _sendMessage(...)
```

The purpose of this function is to send messages from L1 to L2. However, the `callValueRefundAddress` is set as a refund address. This address has a power to cancel the retryable ticket completely, compromising the delivery of messages. The solution is to provide `address(0)` in that field. More details here.

20. `_sendMessage` function (ArbitrumDepositProcessorL1 and WormholeDepositProcessorL1)

Severity: Medium

The following function is implemented in ArbitrumDepositProcessorL1 and WormholeDepositProcessorL1 contracts:

```
function _sendMessage(...)
```

The purpose of this function is to send messages from L1 to L2. However, if the `refundAccount` is provided as a zero address, it is incorrectly defaulted not to the `tx.origin` address, but to a `msg.sender` one, which is the Dispenser contract address. The solution is to revert the function when a zero address is provided for the `refundAccount`. More details here.

21. `retain` function

Severity: Medium

The following function is implemented in the Dispenser contract:

```
function retain() external
```

The purpose of this function is to retain staking inflation by voting for a specific retainer address, such that the retained inflation is returned back to the tokenomics. However, the retainer address must never be used via claiming functions, as the retaining inflation will be incorrectly sent to an artificial retainer address and that inflation will be irrecoverable. The solution is to revert if retainer is among the claimed staking contract addresses. More details here.