Swan
Security Review

OCTOBER, 2024

www.chaindefenders.xyz
https://x.com/ChDefendersEth

Lead Auditors

PeterSR

0x539.eth

Table of Contents

Protocol Summary

Swan is structured like a market, where buyers are AI agents. By setting parameters like backstory, behavior, and objective you define how your agent will act. By using the budget you deposit, the agent buys the best items listed based on how aligned they are with its parameters. With each new asset bought, the agent's state changes and the simulation evolves.

Asset creators on the other side of the market are trying to come up with the best asset for a specific agent so that they can profit from selling it. Each agent has a fee rate where the asset creators pay a % of the listing price as a fee to the agent.

The LLM tasks are completed by a decentralized network of oracle nodes and the decisions are executed onchain. Swan enables human-AI interaction at scale in a financialized context with sustainable economics and creates simulated worlds for any scenario people want.

Disclaimer

The ChainDefenders team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

Likelihood/Impact	High	Medium	Low
High	H	H/M	M
Medium	H/M	M	M/L
Low	M	M/L	L

Audit Details

Scope

Id	Files in scope
1	Statistics.sol
2	LLMOracleCoordinator.sol
3	LLMOracleManager.sol
4	LLMOracleRegistry.sol
5	LLMOracleTask.sol
6	BuyerAgent.sol
7	Swan.sol
8	SwanAsset.sol
9	SwanManager.sol

Roles

Id	Roles
1	Swan Owner (Trusted)
2	BuyerAgent Owner
3	SwanAsset Owner
4	Swan Operator (Trusted)

Executive Summary

Issues found

Severity	Count	Description
High	3	Critical vulnerabilities
Medium	4	Significant risks
Low	3	Minor issues with low impact
Informational	0	Best practices or suggestions

Severity	Count	Description
Gas	0	Optimization opportunities

Findings

High

High 01 Duplicate NFT

Summary

More than one `SwanAsset` with the same combination of name, symbol, description can exist in the system.

Vulnerability Details

In `Swan::list` when creating the asset here:

```
address asset = address(swanAssetFactory.deploy(_name, _symbol, _desc,
    msg.sender));
listings[asset] = AssetListing({
    createdAt: block.timestamp,
    royaltyFee: buyer.royaltyFee(),
    price: _price,
    seller: msg.sender,
    status: AssetStatus.Listed,
    buyer: _buyer,
    round: round
});
```

It is never checked whether another asset with the same name, symbol and description exist. This can lead to two or more assets being identical (but with different prices for example). This can disincentivise users to use the platform to purchase assets as they hold no real value if everyone can list the exactly same asset with a different price.

Impact

Incentive loss for users to use the protocol to purchase assets.

Tools Used

Manual Review

Recommendations

Add a check whether the given combination of name, symbol and description already exists for another asset - if so revert the listing transaction.

High 02 Underflow Problem

Summary

In the `LLMOracleCoordinator` :: `finalizeValidation` function, there is a potential for the function to revert under specific conditions during score calculations. This function is triggered when the final validator completes validation for a request. It iterates over all generators and assigns scores to them based on validator scores. However, there are two lines of code where a reverse can occur:

```
1 if ((score ⩾ _mean - _stddev) && (score ⩽ _mean + _stddev))
2 if (generationScores[g_i] ⩾ mean - generationDeviationFactor * stddev
    )
```

Vulnerability Details

In scenarios such as:

1. Four responders and four validators are present.
2. For the first responder, validator scores are `[1, 1, 5, 100]`.
3. Here, the mean is `26`, and the standard deviation is `48`.
4. The condition `score ⩾ _mean - _stddev` will fail, causing the code to revert.

Impact

This reversion causes the function to fail due to potential underflow or overflow. Request can't be finalized and this will lead to wasted efforts by generators and validators will be. And all the fees will be stuck in the contract.

Tools Used

Manual Review

Recommendations

Make the following code adjustments to handle cases where `_mean` is less than `_stddev`:

```
for (uint256 v_i = 0; v_i < task.parameters.numValidations; ++v_i) {
    uint256 score = scores[v_i];
+    if (_mean > _stddev) {
        if ((score ≥ _mean - _stddev) && (score ≤ _mean + _stddev))
    {
            innerSum += score;
            innerCount++;

            // Send validation fee to the validator
            _increaseAllowance(validations[taskId][v_i].validator,
    task.validatorFee);
        }
+    } else {
+        if (score ≤ _mean + _stddev) {
+            innerSum += score;
+            innerCount++;

+            // Send validation fee to the validator
+            _increaseAllowance(validations[taskId][v_i].validator,
    task.validatorFee);
+        }
    }
}
```

Apply a similar fix to handle the condition `if (generationScores[g_i] ≥ mean - generationDeviationFactor * stddev)`.

High 03 Variance Is Not Working

Summary

The function `Statistics::variance` calculates the variance of a dataset. However, the current implementation is prone to a revert error when attempting to compute the difference between each data point and the mean, particularly for values less than the mean. This happens because of underflow in the subtraction operation.

Vulnerability Details

In the current implementation, when calculating variance, the function iterates over each data point in the array and calculates the difference between the data point and the mean. If a value in data is less than the mean, subtracting mean from that value results in a negative number, which causes an underflow in unsigned integer arithmetic, leading to a revert.

Example Scenario:

Input array: [1, 2, 3, 4, 5]

The mean is calculated as 3.

When the first element 1 is processed, the function attempts to compute 1 - 3, which results in an underflow for uint256 and causes the function to revert.

Impact

This vulnerability effectively breaks the variance function for any input where one or more data points are less than the mean, making the function unreliable and unusable for many typical datasets.

Tools Used

Manual Review

Recommendations

Make the following changes:

```
function variance(uint256[] memory data) internal pure returns (
    uint256 ans, uint256 mean) {
     mean = avg(data);
     uint256 sum = 0;
     for (uint256 i = 0; i < data.length; i++) {
-         uint256 diff = data[i] - mean;
-         sum += diff * diff;
+         int256 diff = data[i] - mean;
+         sum += uint256(diff * diff);
     }
     ans = sum / data.length;
}
```

Medium

Mid 01 Buyer Agent Will Lose Fees In Some Situations

Summary

In the `BuyerAgent::oraclePurchaseRequest` and `BuyerAgent::oracleStateRequest` functions, requests are created and sent to an oracle. However, if these requests are initiated towards the end of a phase, they are likely to remain unfulfilled due to the limited remaining time, resulting in a loss of any fees paid by the buyer.

Vulnerability Details

Consider this scenario:

1. A buyer initiates a purchase request during the Buy phase, but only 10 seconds remain in the phase.
2. Given the short time, the request will likely not be processed in time, causing it to go unfulfilled.
3. As a result, the buyer loses the fees paid to the oracle

Impact

This vulnerability could lead to frequent loss of buyer fees when requests are made near the end of a phase. This could deter users from engaging with the protocol and result in decreased trust, as they may incur unexpected losses due to unfulfilled requests.

Tools Used

Manual Review

Recommendations

Restrict Late Requests: Prevent requests from being created if more than 50% of the current phase time has elapsed. This will reduce the likelihood of unfulfilled requests and prevent unnecessary fees from being charged. Another approach is to set a `minTime` - if the time left in the current phase is less than this `minTime`, the request should revert.

Mid 02 Registry Whitelisting

Summary

In the `LLMOracleRegistry` contract, any account can register as an oracle by staking a specified amount of tokens. However, the contract lacks mechanisms for managing or penalizing malicious oracles. Specifically, once registered, an oracle cannot be forcefully unregistered by the contract owner, even in cases of malicious behaviour. Consequently, there is no way for the owner to penalize such behaviour by seizing the staked tokens or removing the oracle from the registry.

Impact

A malicious oracle in the system can negatively affect the accuracy of responses or validations, potentially providing erroneous data. This impact is especially critical in scenarios where a single oracle serves as the sole validator or responder. This will affect for example the `BuyerAgent` and the purchase function will revert.

Tools Used

Manual Review

Recommendations

There are two ways to handle this problem:

1. Consider implementing a penalty mechanism. If an oracle acts maliciously, the owner can unregister this oracle and get the staked amount.
2. Only the owner can whitelist oracles. This can prevent malicious oracles to register.

Mid 03 Withdraw All Of The Funds

Summary

In the `LLMOracleCoordinator::withdrawPlatformFees` function, the admin has the ability to withdraw all tokens held by the contract. This capability poses significant risks, as the admin could deplete the contract's funds, preventing validators and generators from withdrawing their due rewards.

Vulnerability Details

Consider the following scenario:

1. The contract owner withdraws all token amounts from the contract.
2. A validator attempts to withdraw their earned fees.
3. With no funds left in the contract, the validator is unable to receive their reward for validating generator responses.

Potentially leading to decreased participation and trust in the system.

Impact

The admin's ability to drain the contract can result in validators and responders losing trust in the system because they can't withdraw their funds.

Tools Used

Manual Review

Recommendations

Addressing this vulnerability is crucial, and while the fix is not trivial, a viable solution involves implementing a mechanism to track fees allocated to validators and responders. Here's a proposed modification to the `withdrawPlatformFees` function:

```
function withdrawPlatformFees() public onlyOwner {
    uint256 totalFees = feeToken.balanceOf(address(this));
    require(totalFees > allocatedFees, "Insufficient available fees to
    withdraw");

    feeToken.transfer(owner(), totalFees - allocatedFees);
}
```

This ensures that the owner can only withdraw funds that do not affect the rewards due to validators and responders, preserving the integrity of the contract and its ecosystem.

Mid 04 Sellers Should Be Whitelistable

Summary

In the current implementation of `Swan::list`, sellers can list new `SwanAssets`, but there is no restrictions on who can be a seller. This allows a malicious seller to spam the asset list with dummy assets, potentially filling the buyer's list with low-priced assets that the buyer may be forced to purchase.

Vulnerability Details

Scenario:

1. Suppose a buyer enters a round with a limit of 10 assets, but currently has no assets listed for purchase.

2. A malicious seller could then list dummy `SwanAssets` with prices just below the protocol fee threshold (e.g., a price of 9 if the fee is 10), effectively bypassing fee requirements.
3. This fills the buyer's asset list with low-value dummy assets, which could force the buyer to spend on undesired assets.

Impact

This vulnerability allows malicious sellers to spam the buyer's asset list, potentially leading to:

- Resource waste within the protocol by flooding the listing mechanism.
- High costs for buyers who may be forced to purchase low-value, unwanted assets.
- Increased risk of DoS attacks as spamming limits the availability of valid assets for genuine sellers and buyers.

Tools Used

Manual Review

Recommendations

Implement a seller whitelisting mechanism to ensure only approved sellers can list assets, preventing spam listings and preserving the protocol's integrity. Additionally, include asset validity checks to prevent listings that circumvent protocol fees.

Suggested Implementation

Add whitelisting and validation checks in `Swan::list` as follows:

```
function list(string calldata _name, string calldata _symbol, bytes
    calldata _desc, uint256 _price, address _buyer)
     external
{
     // Ensure only approved sellers can list assets
+    require(isWhitelistedSeller(msg.sender), "Seller is not
    whitelisted");
```

```
6  }
7
8  // Whitelist function for approved sellers
9  +function isWhitelistedSeller(address seller) internal view returns (
       bool) {
10 +    return approvedSellers[seller];
11 +}
```

The same logic should be applied for the `relist` function.

Low

Low 01 Unfair Stake Amount Policy

Summary

Changed stake amounts open the possibility for unfair situations where some oracles have lower/higher stakes.

Vulnerability Details

In `LLMOracleRegistry` oracles can register. For this to happen there is a minimum stake amount that they have to abide by:

- generatorStakeAmount
- validatorStakeAmount

However, the owner of the protocol can change these stake amounts through `setStakeAmounts` function. This opens the possibility for unfair situations where some oracles have lower/higher stakes because when this change is performed there is no check whether the already registered oracles abide by the new value.

Impact

Such vulnerability makes the protocol's logic unfair and disincentivises users to register.

Tools Used

Manual Review

Recommendations

A possible mitigation for such an issue is to remove the function which changes these values and to leave them immutable - they will be set only once when the protocol is deployed.

Low 02 Unfair Generation Of Response When No Validators

Summary

In the `LLMOracleCoordinator` :: `getBestResponse` function, all results from a request are iterated to find the response with the highest score. This function is intended to select the best response based on scores assigned by validators. However, if there are no validators available for a given request, the function defaults to returning the first generated response. This introduces potential bias, as it fails to objectively determine the best response in the absence of validators.

Vulnerability Details

Consider a scenario where:

1. A `BuyerAgent` initiates a request.
2. Ten generators produce responses, and the request is considered complete.
3. When the `BuyerAgent` calls `getBestResponse`, the function will return the first response by default, since no validators exist to evaluate and score the responses.

In this situation, the system fails to apply an unbiased selection process, potentially leading to unintended or suboptimal results for requests with zero validators.

Impact

This flaw can lead to biased and potentially inaccurate response selection for requests without validators. It could create an unfair advantage for responses generated first, regardless of quality. If exploited, malicious generators could submit low-quality responses early, knowing they would be selected in the absence of validation. This bias could erode trust in the system's reliability and quality, especially for critical requests.

Tools Used

Manual Review

Recommendations

To address this issue, implement the following mechanism to ensure fair selection when no validators are available:

1. Randomized Selection in Absence of Validators: On the first invocation of `getBestResponse` for requests without validators, generate a random index to select one of the generated responses.
2. Cache Random Index: Store the generated index to ensure consistency for subsequent calls to `getBestResponse` for the same request. This avoids reselecting a different response upon each invocation and maintains stable results across repeated queries.

By implementing this approach, the function will avoid bias toward the first response and ensure fairer selection, even in validator-free scenarios.

Low 03 Precision Loss Could Lead To DoS

Summary

In `Swan::transferRoyalties` `buyerFee` and `driaFee` will experience precision loss due to the base being 100. On top of that, the value can be 0 and with certain ERC20 tokens a transfer of 0 will revert DoS-ing the listing and relisting functions for such assets.

Vulnerability Details

As we know due to Solidity's nature when a division happens precision loss can occur, rounding down the result. For the `transferRoyalties` function this will mean that `buyerFee` and `driaFee` will be rounded down in most cases. In the case in which:

- `asset.price` * `asset.royaltyFee` < 100
- `buyerFee` * `platformFee` < 100

The fee(s) will be equal to 0. However, some ERC20 tokens will revert on transfer of 0 which will mean that the `transferRoyalties` function will revert.

Impact

Such a vulnerability will cause certain combinations of price, royaltyFee and platformFee to not be possible without a clear explanation.

Tools Used

Manual Review

Recommendations

To fix this issue perform these tasks:

- When `buyerFee` is 0 don't perform any transfers to not cause a random revert.
- Consider changing the base of the fee to a 1e4 or 1e5 base giving more freedom for fee choice and less possibility for precision loss.