AI IQ Protocol Audit Report

PLATFORM: CODE4RENA

Prepared by:OLA Hamid

Table of Contents

Protocol Summary

Protocol does X, Y, Z

Disclaimer

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

Risk Classification

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

I use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

The Agent Tokenization Platform (ATP) is the first platform designed for creating, launching, and managing fully autonomous, tokenized on-chain agents. These agents are intelligent, adaptable, and tailored for specific roles within decentralized ecosystems.

Scope

See scope.txt

Roles

Role	Description
AgentFactory Owner	admin rights

Issues found

Severity	Number of issues found
High	3
Medium	4
Low	0
Info	INFI
Total	7

Findings

[REPORTED] [H-01] Storage Collision Risk in Proxy Implementation Due to Missing Storage Gap

HIGH

Description: The `Agent.sol` contract implemented a proxy pattern but does not enforce a proper storage gap protection between the `Agent.sol` state variables and the upcoming `implementation` contracts. An incompatible implementation can overwrite critical state variables(token, stages, Ownable, etc) due to storage slot collisions in the `Agent.sol` contract, leading to loss of funds or governance control

Impact:

1. Storage variables can be overwritten during upgrades
2. Critical state corruption in proxy pattern
3. Potential economic loss through state manipulation

Proof of Concept:

1. running command `forge inspect Agent storage-layout --pretty` let you see the the different storage slots. for this POC we will be overwriting slot8.
2. create an incompatable implementation contract to overwrite slot8
3. paste the test in the `proxyTest.sol` test contract and run `forge test`

```
contract MaliciousImplementation {
    // This variable is intended to occupy the same storage slot as
Agent.token
    uint256 public dummy0;

    uint256 public dummy1;
    uint256 public dummy2;
    uint256 public dummy3;
    uint256 public dummy4;
    uint256 public dummy5;
    uint256 public dummy6;
    uint256 public dummy7;

    address public maliciousToken;

    function setMaliciousToken(address _token) public {

    maliciousToken = _token;

    }
    function test_ProxyCollisison() public {
        setUpFraxtal(12_918_968);
        AgentFactory factory = new AgentFactory(currencyToken, 0);
        factory.setAgentBytecode(type(Agent).creationCode);
        factory.setGovenerBytecode(type(TokenGovernor).creationCode);

factory.setLiquidityManagerBytecode(type(LiquidityManager).creationCode);
        factory.setTargetCCYLiquidity(1000e18);
        factory.setInitialPrice(0.1e18);
        factory.setMintToAgent(1000); //10%
        factory.setDefaultProxyImplementation(address(new
ProxyContract1()));
        Agent agent = factory.createAgent("AIAgent", "AIA",
"https://example.com", 0);
        AIToken token = agent.token();
        // Deploy the malicious implementation
        MaliciousImplementation maliciousImpl = new
MaliciousImplementation();

        // Whitelist the malicious contract in the factory
        factory.setAllowedProxyImplementation(address(maliciousImpl),
true);

        // Set Agent to stage 1 (alive)
        factory.setAgentStage(address(agent), 1);



        // Verify original token address
        address originalToken = address(agent.token());
        console2.log("Original Token Address:", originalToken);

        vm.startPrank(agent.owner());
        agent.setProxyImplementation(address(maliciousImpl));

           // Call the malicious function to overwrite Agent.token
        (bool success, ) = address(agent).call(
        abi.encodeWithSignature("setMaliciousToken(address)",
address(0xDead))
        );
        require(success, "Call failed");

        // Check if Agent.token is now corrupted
        address newToken = address(agent.token());
        console2.log("Corrupted Token Address:", newToken);
        assert(newToken == address(0xDead));
    }
```

Recommended Mitigation: These are the 2 likely mitigations to be followed:

1. reserve storage slots

```
    contract Agent is ERC721URIStorage, Ownable, Proxy {
+       uint256[50] private __gap;
// .......
    }
```

2. consider using EIP 1967(UUPS) proxy pattarn.

[REPORTED] [H-02] Storage Collision Risk Due to Non-Upgradeable Dependencies

Description: The `Agent.sol` contract inherits from non-upgradeable OpenZeppelin contracts (ERC721URIStorage and Ownable) while acting as a proxy. Non-upgradeable base contracts do not use storage gaps and are not designed for proxy compatibility. Future upgrades to the implementation contract or its dependencies could cause storage layout collisions, corrupting critical state variables.

ROOT CAUSE https://github.com/code-423n4/2025-01-iq-ai/blob/b16b866d4c8d3e4a69b37a02c4e396d4b294537e/src/Agent.sol#L4 **Impact:** If the implementation or its dependencies change their storage layout (e.g., adding/modifying state variables), it could overwrite the proxy's existing storage slots. This could lead to:

Token ownership corruption (e.g., ERC721._owners overwriting Agent.token).

Loss of access control (e.g., Ownable._owner conflicting with Agent.proxyImplementation).

Permanent contract bricking.

Proof of Concept: ***Current Storage Layout:***

```
contract Agent is ERC721URIStorage, Ownable, Proxy {
    // Inherited from ERC721URIStorage (non-upgradeable):
    mapping(uint256 => string) private _tokenURIs; // Slot 0
    // Inherited from Ownable (non-upgradeable):
    address private _owner; // Slot 1
    // Agent's state variables:
```

```
    AIToken public token; // Slot 2
    // ... other variables
}
```

Upgrade Introduces Collision: Suppose ERC721URIStorage is updated in a future OpenZeppelin version to add a new state variable:

```
contract ERC721URIStorage {
    mapping(uint256 => string) private _tokenURIs; // Slot 0
    uint256 public newVariable; // Slot 1 (collides with Ownable._owner!)
}
```

Recommended Mitigation: Use Upgradeable Versions: Replace ERC721URIStorage and Ownable with their upgradeable counterparts:

```
// SPDX-License-Identifier: ISC
pragma solidity >=0.8.25;
// what is the difference between ERC721URIStorage and ERC721
- import {ERC721URIStorage, ERC721} from
"@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
- import {Ownable} from "@openzeppelin/contracts/access/Ownable.sol";
+   import {ERC721URIStorageUpgradeable, ERC721Upgradeable} from
"@openzeppelin/contracts-
upgradeable/token/ERC721/extensions/ERC721URIStorageUpgradeable.sol";
+ import {OwnableUpgradeable} from "@openzeppelin/contracts-
upgradeable/access/OwnableUpgradeable.sol";
```

[REPORTED] [H-3] Deterministic CREATE2 Salt in the `AgentFactory.sol` has a predetermisic `salt` Collision Enables Contract Address Prediction

Description: The Issue here is very similar to `https://github.com/sherlock-audit/2023-12-arcadia-judging/issues/59`. The `deployAgent` and the `deployLiquidityManager` Functions in the `AgentFactory.sol` contract, uses `agents.length` as the salt value for CREATE2 when deploying new Agent contracts. This salt value is entirely predictable as it's simply an incrementing counter that's set to public visibility, callable by anyone. **Description:** The Issue here is very similar to `https://github.com/sherlock-audit/2023-12-arcadia-judging/issues/59`. The `deployAgent` and the `deployLiquidityManager` Functions in the `AgentFactory.sol` contract, uses `agents.length` as the salt value for CREATE2 when deploying new Agent contracts. This salt value is entirely predictable as it's simply an incrementing counter that's set to public visibility, callable by anyone. The deterministic nature of this salt makes it possible for an attacker(specially validators since mempool is private for frax chain) to calculate the addresses of future Agent deployments before they occur.

https://github.com/code-423n4/2025-01-iq-ai/blob/b16b866d4c8d3e4a69b37a02c4e396d4b294537e/src/AgentFactory.sol#L366
https://github.com/code-423n4/2025-01-iq-ai/blob/b16b866d4c8d3e4a69b37a02c4e396d4b294537e/src/AgentFactory.sol#L208

https://github.com/code-423n4/2025-01-iq-ai/blob/b16b866d4c8d3e4a69b37a02c4e396d4b294537e/src/AgentFactory.sol#L159 **Impact:**

1. The major issue here is that Agent and LiquidityManager contract address could be hijacked by a malicious user , which breaks the intended functionality of the protocol.
2. Attacker or a malicious Validator can front-run deployments knowing the exact addresses where contracts will be deployed.
3. The attacker's contract mimics the real LiquidityManager but includes backdoors to steal funds. When the factory calls `moveLiquidity()`, funds could be sent to the attacker's contract instead of Fraxswap.

Root Cause:

```
function deployAgent(
    string memory name,
    string memory symbol,
    string memory url
)
    internal
    returns (Agent agentAddress)
{
@>    uint256 salt = agents.length; // @audit deterministic salt
    bytes memory bytecodeWithArgs = abi.encodePacked(agentBytecode,
abi.encode(name, symbol, url, address(this)));
    assembly {
        agentAddress := create2(0, add(bytecodeWithArgs, 0x20),
mload(bytecodeWithArgs), salt)
        if iszero(extcodesize(agentAddress)) { revert(0, 0) }
    }
}
```

same in the `deployLiquidityManager` Function.

Proof of Concept:

```
function test_Create2Attack() public {
        setUpFraxtal(12_918_968);
        uint256 creationFee = 15e18;
        address whale = 0x00160baF84b3D2014837cc12e838ea399f8b8478;

        // Deploy factory and set bytecode (as before)
        factory = new AgentFactory(currencyToken, 0);
        factory.setAgentBytecode(type(Agent).creationCode);
        factory.setGovenerBytecode(type(TokenGovernor).creationCode);

factory.setLiquidityManagerBytecode(type(LiquidityManager).creationCode);
        factory.setTargetCCYLiquidity(1000e18);
        factory.setInitialPrice(0.1e18);
        factory.setTradingFee(100);
        factory.setCreationFee(creationFee);
        // 1. Calculate future addresses before deployment
        address predictedAgent1 = calculateFutureAgentAddress(0);

        // 2. Deploy first agent
        vm.startPrank(whale);
        currencyToken.approve(address(factory), 115e18); // fee + initial
swap
        Agent agent1 =
factory.createAgent("TestAgent","TEST","https://example.com", 100e18);
        vm.stopPrank();

        assertEq(address(agent1), predictedAgent1, "agent address
misCatch");
    }

    function calculateFutureAgentAddress(uint256 futureLength) public view
returns (address) {
        bytes memory bytecode = abi.encodePacked(
            factory.agentBytecode(),
            abi.encode(
                "TestAgent",
                "TEST",
                "https://example.com",
                address(factory)
            )
        );

        bytes32 salt = bytes32(futureLength);
        bytes32 hash = keccak256(
            abi.encodePacked(
                bytes1(0xff),
                address(factory),
                salt,
                keccak256(bytecode)
            )
        );

        return address(uint160(uint256(hash)));
    }
```

Recommended Mitigation: Use a more unpredictable salt, maybe consier using a nonce increment to it for extra assurance.

```
uint256 salt = uint256(keccak256(abi.encodePacked(
    msg.sender,
    block.timestamp,
    agents.length,
    name,
    symbol
)));
```

MEDIUM

[REPORTED] [M-1] THE `TokenGovernor` CONTRACT SET GOVERNANCE VARIABLES TO TESING VALUES INSTEAD OF PRODUCTION VALUES.

Description: The `TokenGovernor.sol` contract uses testing grade timeLock variables, these variables `TokenGovernor::votingDelayInSeconds`, `TokenGovernor::votingPeriodInSeconds`, `TokenGovernor::proposalThresholdPercentage` are set to 2 minutes, 5 minutes and 1% respectively.If the current suite/codebase goes into production, These variables can be dangerous for real-world governance as a malicious can leverage these short timeLock to rush proposals through without proper review, as it takes a total of 7 minutes before the setter function are called to chenge the variables values. **rootcause** https://github.com/code-423n4/2025-01-iq-ai/blob/b16b866d4c8d3e4a69b37a02c4e396d4b294537e/src/TokenGovernor.sol#L39 https://github.com/code-423n4/2025-01-iq-ai/blob/b16b866d4c8d3e4a69b37a02c4e396d4b294537e/src/TokenGovernor.sol#L38 **Impact:** Governance Attacks: Short voting periods allow malicious actors to push through proposals before proper community review.

Proof of Concept:

```
    // Settings for testing
    uint32 public votingDelayInSeconds = 2 minutes; // 2 minutes in seconds
    uint32 public votingPeriodInSeconds = 5 minutes; // 5 minutes in seconds
    uint32 public proposalThresholdPercentage = 1; // 0.01%
```

Recommended Mitigation:

```
    // The voting delay in seconds
+    uint32 public votingDelayInSeconds = 1 days; // 1 day in seconds

    // The voting period in seconds
+    uint32 public votingPeriodInSeconds = 7 days; // 1 week in seconds

    // The proposal threshold percentage
+    uint32 public proposalThresholdPercentage = 100; // 1% of supply

-    // Settings for testing
-    uint32 public votingDelayInSeconds = 2 minutes; // 2 minutes in seconds
-    uint32 public votingPeriodInSeconds = 5 minutes; // 5 minutes in seconds
-    uint32 public proposalThresholdPercentage = 1; // 0.01%
```

[REPORTED] [M-3] LACK OF DEADLINE/NONCE PARAMETER, LEADS TO STALE PRICE EXECUTION.

Description: The `BootStrapPool::Buy` and `BootStrapPool::Sell` functions are both set to public visibility, if the transaction has not been confirmed for a long time then the user might end up with a token amountOut that is not accurate to the position placed in. Also allowing transactions to execute at outdated prices if delayed in the mempool.

https://github.com/code-423n4/2025-01-iq-ai/blob/b16b866d4c8d3e4a69b37a02c4e396d4b294537e/src/BootstrapPool.sol#L85
https://github.com/code-423n4/2025-01-iq-ai/blob/b16b866d4c8d3e4a69b37a02c4e396d4b294537e/src/BootstrapPool.sol#L105 **Root Cause**

- Missing Deadline: Transactions remain valid indefinitely, allowing execution after price changes.

Impact: Transaction executes after long delays at unexpected prices, Affected Users receive fewer tokens than expected due to price changes caused by malicious actors or market volatility.

Proof of Concept: In the PoC provided,the user's output drops from `10,999,879,001,330,985` tokens to `8,927,758,292,404,228` tokens (18%+ loss) after a whale maliciously or unintentionally manipulate reserves.

```
function testStalePriceExecutionLoss() public {
        _bootstrapSetup();
        LiquidityManager manager =
LiquidityManager(factory.agentManager(address(agent)));
        bootstrapPool = manager.bootstrapPool();

        uint256 amountToBuy = 100e18;
        address attacker = makeAddr("attacker");
        address user = makeAddr("user");

        deal(address(token), address(bootstrapPool), 1000 ether);
        deal(address(currencyToken), attacker, 1000 ether);
        deal(address(token), user, 1000 ether);
        deal(address(currencyToken), user, 1000 ether);


        uint256 initialAmountOut = bootstrapPool.getAmountOut(amountToBuy,
address(currencyToken));
        uint256 currencyPoolAmoout =
currencyToken.balanceOf(address(bootstrapPool));
        uint256 tokenPoolAmount = token.balanceOf(address(bootstrapPool));
        console.log (initialAmountOut, "initialAmountOut");
        console.log (currencyPoolAmoout, "currencyPoolAmoout");
        console.log (tokenPoolAmount, "tokenPoolAmount");

        vm.roll(block.number + 100);
        vm.warp(block.timestamp + 1 hours);

        address whale = makeAddr("whale");
        deal(address(currencyToken), whale, 1_000_000e18);
        vm.startPrank(whale);
        currencyToken.approve(address(bootstrapPool), type(uint256).max);
        bootstrapPool.buy(1_000_000e18, whale); // Drastically reduces
agentToken reserves
        uint256 newCurrencyAmount =
currencyToken.balanceOf(address(bootstrapPool));
        uint256 newTokenAmount = token.balanceOf(address(bootstrapPool));
        console.log (newCurrencyAmount, "newCurrencyAmount");
        console.log (newTokenAmount, "newTokenAmount");

            // User's transaction executes at a worse price
        vm.startPrank(user);
        currencyToken.approve(address(bootstrapPool), amountToBuy);
        uint256 actualAmountOut = bootstrapPool.buy(amountToBuy, user);
        vm.stopPrank();

        console.log (actualAmountOut, "actualAmountOut");

        assertTrue(actualAmountOut < initialAmountOut);
    }
```

Recommended Mitigation: There following mitigation should be considered:

1. changing the `buy`, `sell` an other internal functions handling the logic to internal vsibility. *saves gas*
2. add `_deadline` parameter to the missing `buy` and `sell` functions.

```diff
    function buy(uint256 _amountIn,
    address _recipient
+    uint256 _deadline
    ) public nonReentrant notKilled returns (uint256) {
+        if (block.timestamp > _deadline ) {
+            revert E_deadlinePassed();
        }
        uint256 _amountOut = getAmountOut(_amountIn,
address(currencyToken));
        //......

    }
```

[REPORTED] [M-4] LACK OF `minOutputAmount` CHECK IN BOOTSTRAP POOL LEADS TO EXCESSIVE SLIPPAGE

Description: The `BootStrapPool::Buy` and `BootStrapPool::Sell` functions lack a minimum output amount parameter (`minAmountOut`). despite, this check occured in the `router` contract, the `BootStrapPool::Buy` and `BootStrapPool::Sell` functions were set to `public` visibility allowing this finiding potentially valid. This allows trades to execute with unlimited slippage, potentially resulting in users receiving significantly fewer tokens than expected. https://github.com/code-423n4/2025-01-iq-ai/blob/b16b866d4c8d3e4a69b37a02c4e396d4b294537e/src/BootstrapPool.sol#L85 https://github.com/code-423n4/2025-01-iq-ai/blob/b16b866d4c8d3e4a69b37a02c4e396d4b294537e/src/BootstrapPool.sol#L105 **Root Cause**

- Missing Slippage Protection: No minimum output amount check allows trades to execute at any price, regardless of how unfavorable
- Output amount is calculated at execution time without bounds checking

Impact: Users can receive significantly fewer tokens than expected due to:

- Large trades impacting the price before user's transaction
- Market volatility during block inclusion
- MEV attacks (even with private mempool, validators can still sandwich trades)
- No way for users to specify acceptable price bounds

Proof of Concept: The PoC demonstrates how a user can receive significantly fewer tokens due to price impact from a large trade:

```
function testSlippageAttack() public {
    _bootstrapSetup();
    LiquidityManager manager =
LiquidityManager(factory.agentManager(address(agent)));
    bootstrapPool = manager.bootstrapPool();

    uint256 amountToBuy = 100e18;
    address user = makeAddr("user");
    address whale = makeAddr("whale");

    // Setup initial state
    deal(address(token), address(bootstrapPool), 1000 ether);
    deal(address(currencyToken), user, 1000 ether);
    deal(address(currencyToken), whale, 10000 ether);

    // Get expected output amount
    uint256 expectedOutput = bootstrapPool.getAmountOut(amountToBuy,
address(currencyToken));
    console.log("Expected output:", expectedOutput);

    // Whale makes large trade impacting price
    vm.startPrank(whale);
    currencyToken.approve(address(bootstrapPool), type(uint256).max);
    bootstrapPool.buy(5000e18, whale); // Large trade impacts price
    vm.stopPrank();

    // User trade executes at worse price
    vm.startPrank(user);
    currencyToken.approve(address(bootstrapPool), amountToBuy);
    uint256 actualOutput = bootstrapPool.buy(amountToBuy, user);
    vm.stopPrank();

    console.log("Actual output:", actualOutput);
    console.log("Lost value:", expectedOutput - actualOutput);
```

```
        assertTrue(actualOutput < expectedOutput);
        // User receives ~30% fewer tokens than expected
    }
```

Recommended Mitigation: Add minimum output amount parameter and check:

```
function buy(
        uint256 _amountIn,
        address _recipient,
+       uint256 _minAmountOut
    ) public nonReentrant notKilled returns (uint256) {
        uint256 _amountOut = getAmountOut(_amountIn,
address(currencyToken));
+       if (_amountOut < _minAmountOut) {
+           revert E_InsufficientOutputAmount();
+       }
        currencyTokenFeeEarned += _amountIn - (_amountIn * fee) / 10_000;
        currencyToken.safeTransferFrom(msg.sender, address(this),
_amountIn);
        agentToken.safeTransfer(_recipient, _amountOut);
        emit Swap(msg.sender, _amountIn, 0, 0, _amountOut, _recipient);
        return _amountOut;
    }
```

[REPORTED] [M-5] Ether Locked and Token in `Agent.sol` Contract.

Description: The `Agent.sol` contract has a recieve function that is made available for receiving ether, mostly the only feasible way to receive ether is through an EOA sending direct ether to the contract or through codebase, the `setAgentStage` which is set with agent contract being wrapped `payable`, there is minimu check to the `msg.value` but there is still possibility of fund being sent to the contract and it becomes lost.
also the `MovrLiquidity` function in the `LiquidityManager.sol` contract sends `AgentToken` to the `Agent.sol` contract. which lacks a withdraw function, hence token becomes locked. **Impact:** Ether/Token sent to the contract (intentionally or accidentally) becomes irrecoverable, violating the "no locked funds" principle.

Root Cause

```
// Agent.sol
receive() external payable {} // Accepts ETH but no way to withdraw
```

```
// LiqudityMnager.sol
agentToken.safeTransfer(address(agent),
agentToken.balanceOf(address(this)));
```

Proof of Concept:

```
function test_LockedEther() public {
    address user = makeAddr("user");
    vm.deal(user, 1e18);
    vm.prank(user);
    (bool success, ) = address(agent).call{value: 1e18}("");
    assert(success);
    assertEq(address(agent).balance, 1e18); // ETH is locked
}
```

Recommended Mitigation: Add a withdraw Function for bought ETHER AND TOKEN.

```
+ import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
// ... remaining codes ....
+ function withdrawETH() external onlyOwner {
+     payable(owner()).transfer(address(this).balance);
+ }

+ function withdrawToken(IERC20 token, uint256 amount) external onlyOwner
{
+     uint256 balance = token.balanceOf(address(this));
+     require(amount <= balance, "Insufficient token balance");

+     SafeERC20.safeTransfer(token, owner(), amount);

+     emit TokensWithdrawn(owner(), address(token), amount);
+ }
```