Protocol Audit Report

Prepared by: Bizarro

Table of Contents

Protocol Summary

AIToken

- ERC20 Compliant Token which confers governance over `Agent` Contract.
- Contains ERC20Permit Functionality.
- Token contract is owned by the `Agent` Contract.
- Token Clock based on timestamp rather than blockNumber

Agent

- `Agent` Contract which allows for call forwarding to whitelisted implementation contracts.
- Owner of the `Agent` is the `TokenGovernor` contract.
- Whitelists must be approved via the `AgentFactory` contract.
- Implementations must adhere to the storage layout set forth in `Agent`
- Similar to EIP-897 upgradability pattern

AgentFactory

- Contract responsible for deploying the Agent Contract array
- On `createAgent()` call the factory will deploy several contracts:
    1. `Agent`
    2. `AIToken`
    3. `TokenGovernor`
    4. `LiquidityManager` --`initializeBootstrapPool()`--> 4.1 `BootstrapPool`
- AITokens in this step will be allocated between the `Agent`, `DAO` & `LiquidityManager` at this point.
- Users will have an option to perform an initial buy through the `BootstrapPool` contract on the initial call.

AgentRouter

- Contract used to route trades either buying or selling a given `AIToken`
- Will swap either through the `BootstrapPool` or a `Fraxswap` pair

BootstapPool

- Serves as an initial pool through with an `AIToken` can be traded.
- Owned by `LiquidityManager` contract.
- Very similar to X*Y=K style AMM.

LiquidityManager

- Contract intended to move liquidity between the bootstrap pool and the fraxswap pair given certain conditions are met.

TokenGovernor

- Governance contract based off of OZ `Governor.sol`
- Voting token is `AIToken` Governor address will have ownership rights over the `Agent` contract.

Disclaimer

Bizarro found 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
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

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

Scope

See scope.txt

Roles

Executive Summary

Issues found

Severity	Number of issues found
High	5
Medium	2
Low	0
Info	0
Gas	0
Total	7

Findings

High

[H-1] `TokenGovernor::setProposalThresholdPercentage` can cause risk of DAO Governance Lockout.

Description: The `setProposalThresholdPercentage` function in the governance contract allows updating the proposal threshold percentage but lacks proper validation, leading to a Governance Lockout vulnerability. This issue could allow an attacker or even an unintentional misconfiguration to permanently freeze governance operations.

Impact:

1. If `proposalThresholdPercentage` is set to 10,000 (100%), proposals can no longer be made unless a single entity owns 100% of the governance token supply.
2. A malicious governor could set `proposalThresholdPercentage` to 10,000, effectively disabling all future governance proposals.

Proof of Concept: https://github.com/code-423n4/2025-01-iq-ai/blob/b16b866d4c8d3e4a69b37a02c4e396d4b294537e/src/TokenGovernor.sol#L81

Proof of Code: paste the following test to the `TokenGovernorTest.sol` and run `forge test --mt test_exploitProposalThreshold`

```
    function test_exploitProposalThreshold() public {
        factory.setAgentStage(address(agent), 1);
        address[] memory targets = new address[](1);
        targets[0] = address(governor);
        uint256[] memory values = new uint256[](1);
        bytes[] memory calldatas = new bytes[](1);
        calldatas[0] =
abi.encodeWithSignature("setProposalThresholdPercentage(uint32)", 10000);
        string memory description = "Set the ProposalThresholdPercentage
to 10000";
        vm.startPrank(badActor);
        token.delegate(badActor);
        vm.warp(block.timestamp + 1);
        uint256 nonce = governor.propose(targets, values, calldatas,
description);
        vm.warp(block.timestamp + governor.votingDelay() + 1);
        governor.castVote(nonce, 1);
        vm.warp(block.timestamp + governor.votingPeriod());
        governor.execute(targets, values, calldatas,
keccak256(abi.encodePacked(description)));
        vm.stopPrank();
        assert(governor.proposalThresholdPercentage() == 10000);
    }
```

Recommended Mitigation:

```
function setProposalThresholdPercentage(uint32
_proposalThresholdPercentage) public {
        if (msg.sender != address(this)) revert NotGovernor();
-       if (proposalThresholdPercentage > 1000) revert InvalidThreshold();
+       if (_proposalThresholdPercentage > 1000) revert
InvalidThreshold();
        proposalThresholdPercentage = _proposalThresholdPercentage;
        emit ProposalThresholdSet(_proposalThresholdPercentage);
    }
```

[H-2] Lack of Slippage Protection in `BootstrapPool::buy` and `BootstrapPool::sell`

Description: The `BootstrapPool::buy` and `BootstrapPool::sell` functions facilitate token swaps between currencyToken and agentToken. However, these functions do not include a slippage protection parameter, making them vulnerable to price manipulation.

Impact: Without slippage protection, transactions can be sandwiched between two trades, allowing attackers to manipulate token prices to their advantage, leading to unfair price execution for users.

Proof of Concept:

BootstrapPool::buy Implementation

BootstrapPool::sell Implementation

Recommended Mitigation: Implement a slippage protection parameter that allows users to set an acceptable price range for their transactions, preventing price manipulation and sandwich attacks.

[H-3] Lack of Expiration timestamp when swapping tokens

Description: In Buy and sell functions there is no expiration timestamp parameter to pass.

```
    function buy(uint256 _amountIn, address _recipient) public
nonReentrant notKilled returns (uint256) {}
    function sell(uint256 _amountIn, address _recipient) public
nonReentrant notKilled returns (uint256) {}
```

The transaction can be pending in mempool for a long and the trading activity is very time senstive. Without deadline check, the trade transaction can be executed in a long time after the user submit the transaction, at that time, the trade can be done in a sub-optimal price, which harms user's position.

Impact: Increased Risk of Price Manipulation: Delayed execution may result in users receiving significantly different prices than expected.

Front-running Vulnerability: Attackers could exploit delayed transactions to manipulate the market before execution.

Proof of Concept: 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

Recommended Mitigation: Introduce a deadline parameter in both the buy and sell functions, allowing users to specify the maximum time a transaction remains valid. If the transaction is not executed within this timeframe, it should be automatically reverted.

[H-4] Mismatched Token Decimals in `BootstrapPool.sol`

Description: In `BootstrapPool` contract the `getAmountIn` and `getAmountOut` function calls the `getReserves` function to get the token reserves present in the pool. However, the function does not account for the difference in decimal precisions between the two tokens: If currencyToken has 6 decimals. agentToken has 18 decimals. This mismatch causes the calculations to be performed on values with different scales, leading to incorrect results. For example:

If `_amountIn` is `1_000_000` (1 token in 6 decimals) and `_reserveOut` is `50_000_000_000_000_000_000` (50 tokens in 18 decimals), the formula will produce an incorrect `_amountOut` because the values are not normalized to the same scale.

Impact:

1. Users may receive significantly more or fewer tokens than expected during swaps, leading to financial losses.
2. Attackers could exploit the vulnerability to drain funds from the pool by manipulating swap calculations.

Proof of Concept: https://github.com/code-423n4/2025-01-iq-ai/blob/b16b866d4c8d3e4a69b37a02c4e396d4b294537e/src/BootstrapPool.sol#L133C5-L136C6

1. If the pool has 100 currencyToken(assuming the currencyToken has 6 decimals) in the reserve, `_reserveCurrencyToken = 100e6`
2. And the pool has 50 AIToken(18 decimals) in the reserve, `_reserveAgentToken = 50e18`
3. Call getAmountOut with: `_amountIn` = 1_000_000 (1 token of currencyToken) and `_tokenIn` = address(currencyToken)

Observe the incorrect _amountOut value.

Expected Result

The _amountOut should be calculated correctly, accounting for the difference in decimal precisions.

Actual Result

The _amountOut is incorrect due to mismatched decimals.

Recommended Mitigation:

```
function getReserves() public view returns (uint256 _reserveCurrencyToken,
uint256 _reserveAgentToken) {
-        _reserveCurrencyToken = phantomAmount +
currencyToken.balanceOf(address(this)) - currencyTokenFeeEarned;
-        _reserveAgentToken = agentToken.balanceOf(address(this)) -
agentTokenFeeEarned;
+         _reserveCurrencyToken = phantomAmount +
(currencyToken.balanceOf(address(this)) * (10**(18 -
IERC20Metadata(address(currencyToken)).decimals()))) -
currencyTokenFeeEarned;
+        _reserveAgentToken = (agentToken.balanceOf(address(this)) * (10**
(18 - IERC20Metadata(address(currencyToken)).decimals()))) -
agentTokenFeeEarned;
    }
```

[H-5] `LiquidityManager::moveLiquidity` Function Changes the Liquidity Ratio of the Token Pair.

Description: The `moveLiquidity` function in the `LiquidityManager` contract incorrectly assumes that both `currencyToken` and `agentToken` have 18 decimals. This assumption leads to incorrect calculations when adding liquidity to the `fraxswapFactory`, as the function does not account for the actual decimal precisions of the tokens. Specifically:

The `getPrice` function in `BootstrapPool` assumes `currencyToken` has 18 decimals, but it may have a different decimal precision (e.g., 6 decimals).

The `moveLiquidity` function uses the price returned by `getPrice` to calculate the `liquidityAmount`, which results in an incorrect liquidity ratio if `currencyToken` does not have 18 decimals.

This issue can cause the liquidity ratio of the token pair to deviate from the expected ratio, leading to imbalanced pools and potential financial losses.

Impact:

1. The liquidity added to the fraxswapFactory will not match the expected ratio, leading to an imbalanced pool.
2. Users may receive fewer tokens than expected when swapping, leading to financial losses.
3. Attackers could exploit the imbalanced pool to drain funds or manipulate prices.

Proof of Concept:

https://github.com/code-423n4/2025-01-iq-ai/blob/b16b866d4c8d3e4a69b37a02c4e396d4b294537e/src/BootstrapPool.sol#L125C5-L128C6

```
    function getPrice() external view notKilled returns (uint256 _price) {
        (uint256 _reserveCurrencyToken, uint256 _reserveAgentToken) =
getReserves();
@>        _price = (_reserveCurrencyToken * 1e18) / _reserveAgentToken;
    }
```

https://github.com/code-423n4/2025-01-iq-ai/blob/b16b866d4c8d3e4a69b37a02c4e396d4b294537e/src/LiquidityManager.sol#L103C5-L128C6

```
function moveLiquidity() external {
        require(!bootstrapPool.killed(), "BootstrapPool already killed");
        uint256 price = bootstrapPool.getPrice();
@>      (uint256 _reserveCurrencyToken, ) = bootstrapPool.getReserves();
        _reserveCurrencyToken = _reserveCurrencyToken -
bootstrapPool.phantomAmount();
        uint256 factoryTargetCCYLiquidity =
AgentFactory(owner).targetCCYLiquidity();
        require(
            _reserveCurrencyToken >= targetCCYLiquidity ||
_reserveCurrencyToken >= factoryTargetCCYLiquidity,
            "Bootstrap end-criterion not reached"
        );
        bootstrapPool.kill();

        // Determine liquidity amount to add
        uint256 currencyAmount = currencyToken.balanceOf(address(this));
@>      uint256 liquidityAmount = (currencyAmount * 1e18) / price;

        // Add liquidity to Fraxswap
        IFraxswapPair fraxswapPair =
addLiquidityToFraxswap(liquidityAmount, currencyAmount);

        // Send all remaining tokens to the agent.
        agentToken.safeTransfer(address(agent),
agentToken.balanceOf(address(this)));
        currencyToken.safeTransfer(address(agent),
currencyToken.balanceOf(address(this)));
        emit LiquidityMoved(agent, address(agentToken),
address(fraxswapPair));
        AgentFactory(owner).setAgentStage(agent, 1);
    }
```

Recommended Mitigation: Update the `getPrice` and `moveLiquidity` function to account for the actual decimal precision of `currencyToken`

Medium

[M-1] Contract locks Ether without a withdraw function.

Description: `Agent.sol` has payable fallback function and can accept Ether but lacks a corresponding function to withdraw it, which leads to the Ether being locked in the contract. To resolve this issue, please implement a public or external function that allows for the withdrawal of Ether from the contract.

Proof of Concept: https://github.com/code-423n4/2025-01-iq-ai/blob/b16b866d4c8d3e4a69b37a02c4e396d4b294537e/src/Agent.sol#L74

[M-2] Wrong token transfer in the `AgentFactory::createAgent` function.

Description: In `AgentFactory::createAgent` The function calculates the mintToDAOAmount to transfer the amount to the DAO as written in the docs of the protocol.

https://github.com/code-423n4/2025-01-iq-ai/blob/main/README.md#agentfactory

```
AITokens in this step will be allocated between the Agent, DAO &
LiquidityManager at this point.
```

But the function fails to transfer AITOkens to the DAO instead transfers them to the `AgentFactory` contract.

```
@>        if (mintToDAOAmount > 0) token.safeTransfer(address(this),
mintToDAOAmount);
          if (mintToAgentAmount > 0) token.safeTransfer(address(agent),
mintToAgentAmount);
```

Proof of Concept: https://github.com/code-423n4/2025-01-iq-ai/blob/b16b866d4c8d3e4a69b37a02c4e396d4b294537e/src/AgentFactory.sol#L116

Recommended Mitigation:

```
-         if (mintToDAOAmount > 0) token.safeTransfer(address(this),
mintToDAOAmount);
+         if (mintToDAOAmount > 0) token.safeTransfer(address(governance),
mintToDAOAmount);
          if (mintToAgentAmount > 0) token.safeTransfer(address(agent),
mintToAgentAmount);
```