Security Audit Report for AIAC Agent Pool

Date: April 28, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	amber.ac
Target	AIAC Agent Pool

Version History

Version	Date	Description
1.0	April 28, 2025	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-04-30 03:34:40+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is the code repository [1] of AIAC Agent Pool of amber.ac. This protocol is an investment protocol for users to invest ETH to buy AgentToken and users can also stake AgentToken to get rewards. Note this audit only focuses on the smart contracts in the following directories/files:

- contracts/*

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
AIAC Agent Pool	Version 1	bc5b9450471670e5c88d44384712bf1d7e903b42
	Version 2	9af7e7ba9cda21b09d3618bb3743270e67f44db5

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

[1] https://github.com/chiachih-amber/ia_smart_contract

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

- Reentrancy
- DoS
- Access control
- Data handling and data flow
- Exception handling
- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Permission management
- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we found **sixteen** potential security issues. Besides, we have **six** recommendations and **two** notes.

- High Risk: 2
- Medium Risk: 7
- Low Risk: 7
- Recommendation: 6
- Note: 2

ID	Severity	Description	Category	Status
1	High	Potential DoS due to underflow	DeFi Security	Fixed
2	High	Potential DoS due to non zero WETH in the pool	DeFi Security	Fixed
3	Medium	Potential lock of funds due to inconsistent campaign status check	DeFi Security	Fixed
4	Medium	Lack of investment limit check in function `_processInitialInvestment()`	DeFi Security	Fixed
5	Medium	Potential duplicate rewards withdrawals due to unchanged `merkleRoot`	DeFi Security	Fixed
6	Medium	Lack of restriction on the change of unlock strategy	DeFi Security	Fixed
7	Medium	All LP tokens can be withdrawn through the function `withdrawFunds()`	DeFi Security	Confirmed
8	Medium	Potential lock of service fees due to untimely claim	DeFi Security	Fixed
9	Medium	Incorrect value assignment in function `_updateStageIfNeeded()`	DeFi Security	Fixed
10	Low	Potential user loss due to untimely invocation of `withdraw()`	DeFi Security	Confirmed
11	Low	Lack of validation of the `endTime` in the function `setTimeSettings`	DeFi Security	Fixed
12	Low	Incorrect parameter of function _initAgentTokenAndPool()	DeFi Security	Fixed
13	Low	Unfair token allocation and failed thriving transition due to precision loss in acceleration phase	DeFi Security	Partially Fixed
14	Low	Lack of check on `unlockTime` in the function `addUnlockEvent()`	DeFi Security	Fixed
15	Low	Service fees can not be claimed if the AgentToken are all withdrawn by the custodian	DeFi Security	Confirmed

16	Low	Potential replay attacks across contracts	DeFi Security	Confirmed
17	-	Incorrect error message in the function `withdrawReversed()`	Recommendation	Fixed
18	-	Lack of invoking function `_disableInitializers()`	Recommendation	Fixed
19	-	Unlimited approval in the function `stakeETH()`	Recommendation	Fixed
20	-	Lack of validation of `_seedingGoal` and `_accelerationGoal`	Recommendation	Fixed
21	-	Stake validation mismatch in Agent creation flow	Recommendation	Confirmed
22	-	Redundant code	Recommendation	Partially Fixed
23	-	Signature verification will skip when signer is not set	Note	-
24	-	Potential centralization risks	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Potential DoS due to underflow

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The internal function `_getAgentTokenOut()` calculates the returned amount of `AgentToken` based on the current stage of the pool and the ETH invested. In the Acceleration phase, the logic relies on the variable `investAgentTokenTotal` to compute the amount of remaining tokens to be distributed. However, due to precision loss from rounding down in the `Seeding` phase, it is possible that the total number of `AgentToken` sold (i.e., `investAgentTokenTotal`) is slightly less than the intended `SEEDING_ALLOCATION`, even though the required ETH amount has already been raised to trigger the next phase. When entering the Acceleration phase, this discrepancy may cause the subtraction `investAgentTokenTotal - SEEDING_ALLOCATION` to underflow, leading to incorrect token calculations and potentially a transaction revert.

```
function _getAgentTokenOut(
    Stage currentStage,
    uint256 ethIn
) internal view returns (uint256 acceptedEthIn, uint256 agentTokenOut) {
    if (ethIn == 0) revert InvalidInvestAmount();
    if (ethIn % INVESTMENT_UNIT != 0) revert InvalidInvestmentUnit();
    if (currentStage != Stage.Seeding && currentStage != Stage.Acceleration) {
        revert InvalidStage();
    }
```

```

        if (currentStage == Stage.Seeding) {
            acceptedEthIn = calculateAcceptableETH(ethIn, SEEDING_GOAL);
            agentTokenOut = (acceptedEthIn * SEEDING_ALLOCATION) / SEEDING_GOAL;
        } else if (currentStage == Stage.Acceleration) {
            acceptedEthIn = calculateAcceptableETH(ethIn, SEEDING_GOAL + ACCELERATION_GOAL);

            // calculate virtual ETH
            uint256 accelerationETH = investETHTotal - SEEDING_GOAL;
            uint256 currentVirtualETH = ACCELERATION_VIRTUAL_ETH + accelerationETH;
            uint256 newVirtualETH = currentVirtualETH + acceptedEthIn;

            // calculate virtual Agent Token
            uint256 accelerationAgentToken = investAgentTokenTotal - SEEDING_ALLOCATION;
            uint256 currentVirtualAgentToken = ACCELERATION_VIRTUAL_AGENT_TOKEN -
                accelerationAgentToken;
            uint256 newVirtualAgentToken = (ACCELERATION_VIRTUAL_ETH *
                ACCELERATION_VIRTUAL_AGENT_TOKEN) / newVirtualETH;

            //   newVirtualAgentToken         newVirtualAgentToken
            //    agentTokenOut
            //        investAgentTokenTotal + agentTokenOut
            agentTokenOut = currentVirtualAgentToken - newVirtualAgentToken;


            if (investAgentTokenTotal + agentTokenOut > SEEDING_ALLOCATION + ACCELERATION_ALLOCATION
                ) {
                agentTokenOut = SEEDING_ALLOCATION + ACCELERATION_ALLOCATION - investAgentTokenTotal
                    ;
            }
        }
    }
```

Listing 2.1: contracts/AgentPool.sol

Impact This discrepancy may cause the subtraction `investAgentTokenTotal - SEEDING_ALLOCATION` to underflow, leading to incorrect token calculations and potentially a transaction revert.

Suggestion Revise the logic to make sure that the `investAgentTokenTotal == SEEDING_ALLOCATION` at the end of the `SEEDING` stage.

2.1.2 Potential DoS due to non zero WETH in the pool

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Once an `AgentPool` reaches its `ETH` funding goal, it transitions into the `Thriving` phase. At this stage, the protocol attempts to add liquidity with specified amount of `ETH` and `AgentToken` via the `addLiquidityETH()` function in `Aerodrome's Router` contract. In this function, function `quoteAddLiquidity()` is invoked to calculate how many `LP` tokens should be minted, based on the amounts of both tokens being added. However, the function `quoteAddLiquidity()` requires that both reserves of the token pair in the liquidity pool are non-zero or are zero. Otherwise, the function reverts.

In this case, before the `AgentPool` enters the `Thriving` phase, the attacker can preemptively create a pool via `Aerodrome's` factory contract and donate a small amount of `WETH`, then invoke function `sync()` to update the `WETH` reserve. Since the `AgentToken` reserve remains zero while the `WETH` reserve is not zero, any future call to function `addLiquidityETH()` will revert, effectively preventing the protocol from adding liquidity and halting the `AgentPool`'s progress.

```
    (amountToken, amountETH, liquidity) = _aerodromeRouterContract.addLiquidityETH{value:
        initialETHAmount}(
        address(_agentTokenContract),
        false,
        initialAgentTokenAmount,
        initialAgentTokenAmount,
        initialETHAmount,
        address(this),
        block.timestamp
    );
```

Listing 2.2: contracts/AgentPool.sol

Impact The protocol is unable to add liquidity to the `Aerodrome` pool when entering the `Thriving` stage.

Suggestion Add liquidity to the `Aerodrome` pool directly via the function `mint()`.

2.1.3 Potential lock of funds due to inconsistent campaign status check

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `addTokens()` function allows the team to add additional tokens to a campaign for airdrop. Meanwhile, the unclaimed tokens can be withdrawn via the `withdrawReversed()` function when the airdrop ends. The campaign's status will be set as `Inactive` as well. However, the function `addTokens()` only checks that the campaign status is not `Ended`. This means that the team can still add new tokens after the invocation of function `withdrawReversed()`.

Once new tokens are added in this state, they cannot be withdrawn using the function `withdrawReversed()` again. This is because this function reverts if the status is not `Ended`, and `Inactive` does not satisfy this condition. As a result, the added tokens become stuck in the contract unless the team uses the privileged `setActive()` function to reset the campaign status, which introduces an unnecessary dependency on privileged access.

```
function addTokens(uint256 additionalAmount) external onlyRouter {
    if (additionalAmount == 0) revert InvalidAmount();

    //
    CampaignStatus status = this.getStatus();

```

```
    //
    if (status == CampaignStatus.Ended) revert CampaignEnded();

    //
    _campaign.totalAmount += additionalAmount;

    //
    emit TokensAdded(
        _campaign.token,
        additionalAmount,
        _campaign.totalAmount
    );
}
```

Listing 2.3: contracts/AgentAirdropCampaign.sol

Impact This issue may cause tokens to be permanently locked in the contract, or force the team to rely on privileged functions to recover them.

Suggestion The implementation should explicitly prevent adding new tokens once the unclaimed tokens are withdrawn.

2.1.4 Lack of investment limit check in function `_processInitialInvestment()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description According to the protocol design, the launch of an `AgentToken` consists of two phases: `SEEDING` and `ACCELERATION`. Each phase has a predefined ETH fundraising cap. If the ETH raised in the current phase exceeds the cap, the protocol transitions to the next phase. To promote a fair launch, the protocol sets a per-user purchase cap for each phase and they are `SEEDING_MAX_INDIVIDUAL_INVEST` and `ACCELERATION_MAX_INDIVIDUAL_INVEST,` respectively. However, in the `_processInitialInvestment()` function, there is no check to ensure that the user's ethIn is less than `SEEDING_MAX_INDIVIDUAL_INVEST`, which is inconsistent with the intended design.

```
function _processInitialInvestment(
    address _initialInvestor,
    uint256 ethIn
) private returns (uint256 initEthIn, uint256 initAgentTokenOut) {
    if (ethIn % _agentConfigContract.getInvestmentUnit() != 0) revert InvalidInvestAmount();

    (uint256 acceptedEthIn, uint256 agentTokenOut) = _getAgentTokenOut(Stage.Seeding, ethIn);
    if (acceptedEthIn != ethIn) revert InvestRefused();
    if (agentTokenOut == 0) revert InvalidAgentTokenOut();

    _agentTokenContract.transferAndLockSeeding(_initialInvestor, agentTokenOut);
    _recordInvestment(Stage.Seeding, ethIn, agentTokenOut, _initialInvestor);

    return (ethIn, agentTokenOut);
```

```
    }
```

Listing 2.4: contracts/AgentPool.sol

Impact During the initial investment, a user's ethIn input may exceed the per-user purchase cap, which is unfair to other participants and deviates from the intended protocol behavior.

Suggestion Add a check to ensure that the input ethIn is less than the `SEEDING_MAX_INDIVIDUAL_INVEST`.

2.1.5 Potential duplicate rewards withdrawals due to unchanged `merkleRoot`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `AgentStaking`, users can claim rewards by verifying the current `merkleRoot` with `merkleProofs`. Meanwhile, the function `withdraw()` will compare the user's last withdrawn epoch with the `currentUnlockEpoch` to prevent users from repeatedly withdrawing rewards in the same epoch.

However, if the oracle role updates the `currentUnlockEpoch` with `currentUnlockEpoch + 1`, but keeps the `currentMerkleRoot` unchanged, then the users can claim rewards repeatedly with the old `merkleProof` and circumvent the repeated check.

```
function startUnlock(
    bytes32 merkleRoot
) external override onlyOracle {
    currentUnlockEpoch = currentUnlockEpoch + 1;
    currentMerkleRoot = merkleRoot;

    emit UnlockStarted(address(agentToken), currentUnlockEpoch);
}
```

Listing 2.5: contracts/AgentStaking.sol

```
function withdraw(
    uint256 unlockedAmount,
    uint256 rewardsAmount,
    bytes32[] calldata merkleProof
) external override {
    if (!enableWithdraw) revert WithdrawDisabled();
    if (unlockedAmount == 0 && rewardsAmount == 0) revert InvalidAmount("unlockedAmount and
        rewardsAmount");
    if (currentMerkleRoot == bytes32(0)) revert EpochNotStarted(currentUnlockEpoch);

    //
    if (_lastWithdrawnEpoch[msg.sender] >= currentUnlockEpoch) revert AlreadyWithdrawn();

    //  Merkle
    bytes32 leaf = keccak256(abi.encodePacked(msg.sender, unlockedAmount, rewardsAmount));
    if (!MerkleProof.verify(merkleProof, currentMerkleRoot, leaf)) revert InvalidMerkleProof();

```

```
    //
    if (unlockedAmount > userStakedAmount[msg.sender]) revert InvalidAmount("unlockedAmount");
    if (rewardsAmount > userStakedAmount[msg.sender]) revert InvalidAmount("rewardsAmount");

    //     -
    userStakedAmount[msg.sender] -= unlockedAmount;
    //
    _lastWithdrawnEpoch[msg.sender] = currentUnlockEpoch;

    //  AgentToken
    agentToken.safeTransfer(msg.sender, unlockedAmount + rewardsAmount);

    emit Withdrawn(address(agentToken), msg.sender, currentUnlockEpoch, unlockedAmount,
        rewardsAmount);
}
```

Listing 2.6: contracts/AgentStaking.sol

Impact The users can claim rewards repeatedly.

Suggestion Add check in the function `startUnlock()` to prevent the case that the `currentMerkleRoot` remains unchanged in the subsequent epoch.

2.1.6 Lack of restriction on the change of unlock strategy

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `setUnlockStrategy()` function allows the owner to update the unlock strategy of an `AgentToken`, which defines how locked tokens are released to users over time. However, the function does not restrict updates of the unlock strategy after the unlocking has already begun. Changing the unlock strategy mid-way can lead to inconsistencies in the unlocking state, break the expected unlocking logic, and compromise fairness.

```
function setUnlockStrategy(address _unlockStrategy) external onlyOwner {
    if (_unlockStrategy == address(0)) revert InvalidUnlockStrategy();
    unlockStrategy = IUnlockStrategy(_unlockStrategy);
}
```

Listing 2.7: contracts/AgentToken.sol

Impact Incorrect amounts of `AgentToken` can be released due to inconsistent unlock logic.

Suggestion Disallow updates to the unlock strategy once unlocking has started.

2.1.7 All LP tokens can be withdrawn through the function `withdrawFunds()`

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description Once the SEEDING and ACCELERATION phases are passed, the protocol invokes the _addLiquidityToAerodromePool() function to supply a portion of the ETH and AgentToken to the Aerodrome pool. The amount added is determined by the protocol, and the minted LP tokens are sent back to the current contract.

At the same time, the contract includes a withdrawFunds() function, which allows an account with Custodian privileges to withdraw native tokens or ERC20 tokens from the contract. That means the received LP tokens can be withdrawn by Custodian.

```
    (amountToken, amountETH, liquidity) = _aerodromeRouterContract.addLiquidityETH{value: initialETHAmount}(
        address(_agentTokenContract),
        false,
        initialAgentTokenAmount,
        initialAgentTokenAmount,
        initialETHAmount,
        address(this),
        block.timestamp
    );
```

Listing 2.8: contracts/AgentPool.sol

```
function withdrawFunds(
    address recipient,
    address tokenAddress,
    uint256 tokenAmount
) external override nonReentrant onlyCustodian {
    if (recipient == address(0)) revert InvalidRecipient();

    Stage currentStage = _getCurrentStage();
    if (currentStage != Stage.Thriving) revert NotInThrivingStage();

    if (tokenAddress == address(0)) {
        if (address(this).balance < tokenAmount) revert InsufficientBalance();
        (bool success, ) = recipient.call{value: tokenAmount}("");
        if (!success) revert("ETH transfer failed");
    } else {
        IERC20 token = IERC20(tokenAddress);
        if (token.balanceOf(address(this)) < tokenAmount) revert InsufficientBalance();
        token.safeTransfer(recipient, tokenAmount);
    }

    emit FundsWithdrawn(
        address(_agentTokenContract),
        _custodian,
        recipient,
        tokenAddress,
        tokenAmount
    );
}
```

Listing 2.9: contracts/AgentPool.sol

Impact The `withdrawFunds()` function lacks a clearly defined use case, and since the initial liquidity added to the Aerodrome pool can be withdrawn, this may result in potential user losses.

Suggestion Revise the logic to ensure that the initial liquidity added to the pool cannot be withdrawn.

Feedback from the project This is an expected behavior. The custodian is permitted to withdraw LP tokens.

2.1.8 Potential lock of service fees due to untimely claim

Severity Medium

Status Fixed in Version 2

Introduced by Version 1

Description The service fee in the `AgentPool` is calculated with the `periodCountElapsed` and the `AGENT_TOKEN_TOTAL_SUPPLY`. Thus, the service fee will accumulate as the time goes by. If the service fee is not claimed in time, the accumulated fee may exceed the current `AgentToken` balance of the `AgentPool`. In this case, the claiming operation of the service fee will revert due to the insufficient balance. This will finally cause the service fee to be locked in the contract.

```
function _pendingServiceFee(
    Stage currentStage
) internal view returns (
    uint256 serviceFee,
    uint256 servicePeriodStart,
    uint256 servicePeriodEnd
) {
    uint256 blockTimestamp = block.timestamp;
    if (currentStage != Stage.Thriving) {
        return (0, 0, 0);
    }

    servicePeriodStart = serviceFeeClaimTime;
    if (serviceFeeClaimTime == 0) {
        servicePeriodStart = stagePeriods[Stage.Thriving].startTime;
    }

    uint256 timeElapsed = blockTimestamp - servicePeriodStart;
    uint256 periodCountElapsed = timeElapsed / serviceFeePeriod();
    if (periodCountElapsed == 0) {
        return (0, 0, 0);
    }
    else {
        serviceFee = (AGENT_TOKEN_TOTAL_SUPPLY * serviceFeeRate()) / 1_000_000 *
            periodCountElapsed;
        servicePeriodEnd = servicePeriodStart + periodCountElapsed * serviceFeePeriod();
        return (serviceFee, servicePeriodStart, servicePeriodEnd);
    }
}
```

Listing 2.10: contracts/AgentPool.sol

Impact This will finally cause the service fee to be locked in the contract.

Suggestion Add a logic to claim the minimum value of the service fee and the `AgentToken` balance in the function `claimServiceFee()`.

2.1.9 Incorrect value assignment in function `_updateStageIfNeeded()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `AgentPool` contract, the `_updateStageIfNeeded()` function is responsible for updating the stage of an `AgentToken`. Specifically, if the expected amount of ETH is raised within the designated timeframe, the token progresses to the next stage until it is fully launched. However, within the `currentStage == Stage.Seeding` branch, there is an incorrect assignment. The failure time for the Acceleration stage should be set to `stagePeriods[Stage.Acceleration].startTime + STAGE_DURATION_ACCELERATION`, rather than `stagePeriods[Stage.Seeding].startTime + STAGE_DURATION_SEEDING`.

```
            stagePeriods[Stage.Failed].startTime = stagePeriods[Stage.Seeding].startTime +
                STAGE_DURATION_SEEDING;
```

Listing 2.11: contracts/AgentPool.sol

Impact The `AgentToken` may prematurely enter the Failed phase of the Acceleration stage, which is inconsistent with the intended behavior.

Suggestion Revise the logic to ensure the variable is assigned correctly.

2.1.10 Potential user loss due to untimely invocation of `withdraw()`

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the `AgentStaking` contract, an account with `Oracle` privileges can invoke the `startUnlock()` function to set a new epoch and assign the corresponding `currentMerkleRoot`. During each epoch, users can withdraw their deposited assets and rewards by invoking the `withdraw()` function with a valid `merkleProof`. However, if a user fails to withdraw within the designated epoch, their `merkleProof` becomes invalid in the subsequent epoch, as a new `merkleRoot` is set with each update. In this case, the user may suffer a loss.

```
function withdraw(
    uint256 unlockedAmount,
    uint256 rewardsAmount,
    bytes32[] calldata merkleProof
) external override {
    if (!enableWithdraw) revert WithdrawDisabled();
    if (unlockedAmount == 0 && rewardsAmount == 0) revert InvalidAmount("unlockedAmount and
        rewardsAmount");
```

```
        if (currentMerkleRoot == bytes32(0)) revert EpochNotStarted(currentUnlockEpoch);

        //
        if (_lastWithdrawnEpoch[msg.sender] >= currentUnlockEpoch) revert AlreadyWithdrawn();

        //   Merkle
        bytes32 leaf = keccak256(abi.encodePacked(msg.sender, unlockedAmount, rewardsAmount));
        if (!MerkleProof.verify(merkleProof, currentMerkleRoot, leaf)) revert InvalidMerkleProof();

        //
        if (unlockedAmount > userStakedAmount[msg.sender]) revert InvalidAmount("unlockedAmount");
        if (rewardsAmount > userStakedAmount[msg.sender]) revert InvalidAmount("rewardsAmount");

        //      -
        userStakedAmount[msg.sender] -= unlockedAmount;
        //
        _lastWithdrawnEpoch[msg.sender] = currentUnlockEpoch;

        //   AgentToken
        agentToken.safeTransfer(msg.sender, unlockedAmount + rewardsAmount);

        emit Withdrawn(address(agentToken), msg.sender, currentUnlockEpoch, unlockedAmount,
            rewardsAmount);
    }

    /// @inheritdoc IAgentStaking
    function startUnlock(
        bytes32 merkleRoot
    ) external override onlyOracle {
        currentUnlockEpoch = currentUnlockEpoch + 1;
        currentMerkleRoot = merkleRoot;

        emit UnlockStarted(address(agentToken), currentUnlockEpoch);
    }
```

Listing 2.12: contracts/AgentStaking.sol

Impact Users may incur asset losses.

Suggestion Revise the logic to record all previously set `currentMerkleRoot` values, ensuring that users' assets will not be lost.

Feedback from the project This is an expected behavior. The unclaimed amount is calculated off-chain and carried over to the next epoch.

2.1.11 Lack of validation of the `endTime` in the function `setTimeSettings`

Severity Low

Status Fixed in `Version`

Introduced by `Version 1`

Description The `setTimeSettings()` function allows the privileged owner to update the start and end timestamps of a campaign airdrop. However, the function does not validate whether the new end time is in the future. As a result, the owner could set a end time that is already in the past, potentially causing unexpected airdrop behavior.

```
function setTimeSettings(
    uint256 startTime,
    uint256 endTime
) external onlyRouter {
    //
    if (startTime >= endTime) revert InvalidTimeSettings();

    //
    _campaign.startTime = startTime;
    _campaign.endTime = endTime;

    emit TimeSettingsUpdated(
        startTime,
        endTime
    );
}
```

Listing 2.13: contracts/AgentAirdropCampaign.sol

Impact Allows misconfigured or manipulated campaigns that may confuse users or disrupt airdrop timing logic.

Suggestion Add a check to ensure the end time is greater than the current block timestamp.

2.1.12 Incorrect parameter of function _initAgentTokenAndPool()

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `AgentFactory`, the function `_initAgentTokenAndPool()` initializes the `OracleBasedUnlockStrategy` with the oracle parameter to be the `params.owner`. This is incorrect because the owner does not have the oracle logic.

```
    } else if (unlockStrategy == UnlockStrategyType.OracleBased) {
        OracleBasedUnlockStrategy(params.unlockStrategyAddress).initialize(
            params.owner,
            params.agentToken,
            params.owner
        );
```

Listing 2.14: contracts/AgentFactory.sol

```
function initialize(
    address _oracle,
    address _agentToken,
    address _owner
) public initializer {
    if (_oracle == address(0)) revert InvalidAddress("oracle");
    if (_agentToken == address(0)) revert InvalidAddress("token");
```

```

        __Ownable_init();
        _transferOwnership(_owner);

        oracle = _oracle;
        agentToken = _agentToken;
    }
```

Listing 2.15: contracts/unlock/OracleBasedUnlockStrategy.sol

Impact The `OracleBasedUnlockStrategy's` functionality, which can only be called by the oracle address, may not work as expected.

Suggestion Initialize the `OracleBasedUnlockStrategy` with a correct oracle address.

2.1.13 Unfair token allocation and failed thriving transition due to precision loss in acceleration phase

Severity Low

Status Partially Fixed

Introduced by `Version 1`

Description In the `Acceleration` phase of function `_getAgentTokenOut()`, the calculation of `agentTokenOut` relies on the value of `newVirtualAgentToken`, which is computed using a division operation that may suffer from precision loss due to truncation. As a result, `newVirtualAgentToken` can be slightly underestimated, causing `agentTokenOut` to be slightly larger than it should be.

This leads to two issues. First, it is unfair to the last participant. The users who invest earlier will receive more `AgentToken` than expected while the last user may receive less. Second, the `AgentToken` may be sold out before the required ETH amount is fully raised. When subsequent users try to invest, the calculated `agentTokenOut` becomes zero, which is not allowed by the protocol and results in a transaction revert. Since ETH contributions are no longer accepted, the pool can never transition to the `Thriving` phase.

```
            uint256 newVirtualAgentToken = (ACCELERATION_VIRTUAL_ETH * ACCELERATION_VIRTUAL_AGENT_TOKEN) / newVirtualETH;

            //   newVirtualAgentToken         newVirtualAgentToken
            //    agentTokenOut
            //        investAgentTokenTotal + agentTokenOut
            agentTokenOut = currentVirtualAgentToken - newVirtualAgentToken;


            if (investAgentTokenTotal + agentTokenOut > SEEDING_ALLOCATION + ACCELERATION_ALLOCATION) {
                agentTokenOut = SEEDING_ALLOCATION + ACCELERATION_ALLOCATION - investAgentTokenTotal;
            }
```

Listing 2.16: contracts/AgentPool.sol

Impact This issue can result in unfair token distribution and potentially block the pool from moving to the next stage.

Suggestion Round the result up when calculating the value of `newVirtualAgentToken`.

Feedback from the project The DoS vulnerability has been addressed. The price impact is considered negligible and can be safely ignored.

2.1.14 Lack of check on `unlockTime` in the function `addUnlockEvent()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `OracleBasedUnlockStrategy`, an account with `Oracle` privileges can invoke `addUnlockEvent()` to set the unlock time and unlock ratio for the `AgentToken`. If the token's strategy is `OracleBased`, the unlock process relies on these `unlockEvents`. However, the `addUnlockEvent()` function does not check whether the provided `unlockTime` is greater than `unlockStartTime`. Otherwise, it may lead to premature unlocking of the `AgentToken`, which contradicts the intended protocol design.

```
function addUnlockEvent(
    uint256 unlockTime,
    uint256 seedingUnlockRatio,
    uint256 accelerationUnlockRatio
) external onlyOracle {
    if (unlockStartTime == 0) revert UnlockNotStarted();

    //       100%
    if (totalSeedingUnlockRatio + seedingUnlockRatio > 1_000_000) revert UnlockRatioTooHigh("
        seeding");
    if (totalAccelerationUnlockRatio + accelerationUnlockRatio > 1_000_000) revert
        UnlockRatioTooHigh("acceleration");

    //
    totalSeedingUnlockRatio += seedingUnlockRatio;
    totalAccelerationUnlockRatio += accelerationUnlockRatio;

    //
    _insertUnlockEvent(UnlockEvent({
        unlockTime: unlockTime,
        seedingUnlockRatio: seedingUnlockRatio,
        accelerationUnlockRatio: accelerationUnlockRatio
    }));

    emit UnlockEventAdded(
        unlockTime,
        seedingUnlockRatio,
        accelerationUnlockRatio
    );
}
```

Listing 2.17: contracts/unlock/OracleBasedUnlockStrategy.sol

Impact The AgentToken may be unlocked prematurely, which is inconsistent with the intended protocol design.

Suggestion Add a check to ensure that the provided unlockTime must be greater than unlockStartTime.

2.1.15 Service fees can not be claimed if the AgentToken are all withdrawn by the custodian

Severity Low

Status Confirmed

Introduced by Version 1

Description Once the SEEDING and ACCELERATION phases are passed, the protocol can charge the service fee with AgentToken.

At the same time, the contract includes a withdrawFunds() function, which allows an account with Custodian privileges to withdraw native tokens or ERC20 tokens from the contract. This means all the AgentToken in the AgentPool can be withdrawn. In that way, the service fee can not be charged.

```
function claimServiceFee() external {
    Stage currentStage = _getCurrentStage();
    (
        uint256 serviceFee,
        uint256 servicePeriodStart,
        uint256 servicePeriodEnd
    ) = _pendingServiceFee(currentStage);
    if (serviceFee == 0) revert NoServiceFeeAvailable();
    if (block.timestamp < servicePeriodEnd) revert NoServiceFeeAvailable();

    address receipient = feeReceipient();
    _agentTokenContract.safeTransfer(receipient, serviceFee);
    serviceFeeClaimTime = servicePeriodEnd;

    emit ServiceFeeClaimed(
        address(_agentTokenContract),
        receipient,
        serviceFee,
        servicePeriodStart,
        servicePeriodEnd
    );
}
```

Listing 2.18: contracts/AgentPool.sol

Impact In that way, the service fee can not be charged.

Suggestion Revise the logic accordingly.

Feedback from the project The Agent Platform charges service fees to the Agent Project. If the platform cannot collect sufficient fees from the pool, it may terminate the project's agent service.

2.1.16 Potential replay attacks across contracts

Severity Low

Status Confirmed

Introduced by Version 1

Description The `verifyCreateAgentSignature()` and `verifyInvestSignature()` functions generate the message hash for signature verification using a combination of parameters such as the user address, token details, nonce, deadline, and `block.chainid`. However, neither function includes the current contract address (i.e., `address(this)`) in the hash. This omission can lead to replay attack scenarios where the same signature, valid for one contract, could be reused on another contract with identical logic and parameters. Including `address(this)` in the message ensures the signature is bound to the specific contract instance, which is critical for maintaining proper access control in multi-contract deployments.

```
function verifyCreateAgentSignature(
    address creator,
    string memory name,
    string memory symbol,
    uint256 nonce,
    uint256 deadline,
    bytes memory signature,
    address expectedSigner
) public view returns (bool) {
    bytes32 messageHash = keccak256(abi.encodePacked(
        keccak256(abi.encodePacked(
            creator,
            name,
            symbol,
            nonce,
            deadline,
            block.chainid
        ))
    ));

    return _verifySignature(messageHash, signature, expectedSigner);
}

/// @notice
/// @param investor
/// @param agentToken
/// @param stage
/// @param deadline
/// @param signature
/// @param expectedSigner
/// @return
function verifyInvestSignature(
    address investor,
    address agentToken,
    IAgentPool.Stage stage,
    uint256 deadline,
```

```
        bytes memory signature,
        address expectedSigner
    ) public view returns (bool) {
        bytes32 messageHash = keccak256(abi.encodePacked(
            keccak256(abi.encodePacked(
                investor,
                agentToken,
                uint8(stage),
                deadline,
                block.chainid
            ))
        ));

        return _verifySignature(messageHash, signature, expectedSigner);
    }
```

Listing 2.19: contracts/AgentRouter.sol

Impact This issue allows a valid signature to be reused across different contracts.

Suggestion Include `address(this)` in the message hash when computing the digest for signature validation.

Feedback from the project The impact is minor.

2.2 Recommendation

2.2.1 Incorrect error message in the function `withdrawReversed()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The error `InvalidTokenAddress()` used in the function `withdrawReversed()` in both the contracts `AgentAirdropCampaign` and `AgentAirdropRouter` is incorrect since the recipient is not a token address but a user address.

```
    function withdrawReversed(address recipient) external onlyRouter {
        if (recipient == address(0)) revert InvalidTokenAddress();

        //
        CampaignStatus status = this.getStatus();

        //
        if (status != CampaignStatus.Ended)
            revert CampaignNotEnded();

        //
        uint256 unclaimedAmount = _campaign.totalAmount - _campaign.claimedAmount;
        if (unclaimedAmount == 0) revert InvalidAmount();

        //    -
        _campaign.active = false;

```

```
    //
    IERC20(_campaign.token).safeTransfer(recipient, unclaimedAmount);

    emit WithdrawnReversed(
        _campaign.token,
        unclaimedAmount,
        recipient
    );
}
```

Listing 2.20: contracts/AgentAirdropCampaign.sol

```
function withdrawReversed(uint256 campaignId, address recipient) external onlyOwner {
    if (campaignId >= _campaignAddresses.length) revert IAgentAirdropTypes.InvalidCampaignId();
    if (recipient == address(0)) revert IAgentAirdropTypes.InvalidTokenAddress();

    address campaignAddress = _campaignAddresses[campaignId];

    //
    IAgentAirdropCampaign(campaignAddress).withdrawReversed(recipient);
}
```

Listing 2.21: contracts/AgentAirdropRouter.sol

Suggestion Revise the logic accordingly.

2.2.2 Lack of invoking function `_disableInitializers()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `AgentToken` the function `_disableInitializers()` is not invoked in the constructor. Invoking this function prevents the contract itself from being initialized, thereby avoiding unexpected behaviors.

```
contract AgentToken is Initializable, ERC20Upgradeable, OwnableUpgradeable, IAgentToken {
    /// @dev
    struct LockedAmount {
        uint256 seedingLockedAmount;
        uint256 accelerationLockedAmount;
    }

    /// @dev
    address public agentPool;

    /// @dev
    IUnlockStrategy public unlockStrategy;

    /// @dev
    mapping(address => LockedAmount) private _originLockedAmounts;

    /// @dev
    modifier onlyPool() {
```

```
        if (msg.sender != agentPool) revert NotAuthorized();
        _;
    }

    /// @inheritdoc IAgentToken
    function initialize(
        string memory _name,
        string memory _symbol,
        address _agentConfig,
        address _agentPool,
        address _unlockStrategy,
        address _owner
    ) public initializer {
        if (_agentPool == address(0)) revert InvalidPoolAddress();

        __ERC20_init(_name, _symbol);
        __Ownable_init();
        _transferOwnership(_owner);

        agentPool = _agentPool;

        if (_unlockStrategy == address(0)) revert InvalidUnlockStrategy();
        unlockStrategy = IUnlockStrategy(_unlockStrategy);

        uint256 initialSupply = IAgentConfig(_agentConfig).getAgentTokenTotalSupply();
        _mint(address(agentPool), initialSupply);
    }
```

Listing 2.22: contracts/AgentToken.sol

Suggestion Invoke the function `_disableInitializers()` in the constructor to prevent the implementation contract from being initialized.

2.2.3 Unlimited approval in the function `stakeETH()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `AgentStakingProxy` contract, the `stakeETH()` function first converts the user's ETH into `AgentToken`, then stakes the `AgentToken` in the `agentStakingContract` to earn rewards. However, the contract sets the approval amount for `agentStakingContract` to `type(uint256).max`, which is not aligned with best security practices. This approach increases the risk of unintended token transfers and should be replaced with a more restrictive approval strategy.

```
        agentToken.approve(address(agentStakingContract), type(uint256).max);
```

Listing 2.23: contracts/AgentStakingProxy.sol

Suggestion Revise the logic to ensure that the approved amount matches the actual amount of `AgentToken` received from the conversion.

2.2.4 Lack of validation of `_seedingGoal` and `_accelerationGoal`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The protocol requires users to invest in the `AgentPool` with an invested ETH amount to be multiple of `_investmentUnit`. Meanwhile, to achieve the goal of the stage seeding and acceleration, the ETH amount total invested should be equal to the value of the `seedingGoal` and the `accelerationGoal`. Thus it implies that the `_seedingGoal` and `_accelerationGoal` should be divisible by `_investmentUnit`. However, the current logic in the `AgentConfig` does not check this requirement. It may lead to DoS when the parameters in the `AgentConfig` are not correctly set.

```
        _investmentUnit = _params.investmentUnit;

        _stageDurationGenesisSucceeded = _params.stageDurationGenesisSucceeded;
        _stageDurationSeeding = _params.stageDurationSeeding;
        _stageDurationSeedingSucceeded = _params.stageDurationSeedingSucceeded;
        _stageDurationAcceleration = _params.stageDurationAcceleration;

        _seedingGoal = _params.seedingGoal;
        _seedingMinInvestors = _params.seedingMinInvestors;
        _seedingMaxIndividualInvest = _seedingGoal / _seedingMinInvestors;
        _seedingAllocationPercentage = _params.seedingAllocationPercentage;

        _accelerationGoal = _params.accelerationGoal;
```

Listing 2.24: contracts/AgentConfig.sol

```
    function setSeedingGoal(uint256 goal) external override onlyOwner {
        uint256 oldValue = _seedingGoal;
        uint256 oldMaxIndividualInvest = _seedingMaxIndividualInvest;
        _seedingGoal = goal;
        _seedingMaxIndividualInvest = _seedingGoal / _seedingMinInvestors;
        emit ConfigUpdated("seedingGoal", oldValue, goal);
        emit ConfigUpdated("seedingMaxIndividualInvest", oldMaxIndividualInvest,
            _seedingMaxIndividualInvest);
    }
```

Listing 2.25: contracts/AgentConfig.sol

```
    function setAccelerationGoal(uint256 goal) external override onlyOwner {
        uint256 oldValue = _accelerationGoal;
        uint256 oldMaxIndividualInvest = _accelerationMaxIndividualInvest;
        _accelerationGoal = goal;
        _accelerationMaxIndividualInvest = _accelerationGoal / _accelerationMinInvestors;
        emit ConfigUpdated("accelerationGoal", oldValue, goal);
        emit ConfigUpdated("accelerationMaxIndividualInvest", oldMaxIndividualInvest,
            _accelerationMaxIndividualInvest);
    }
```

Listing 2.26: contracts/AgentConfig.sol

Suggestion Add validation of `_seedingGoal` and `_accelerationGoal` to ensure they are divisible by `_investmentUnit`.

2.2.5 Stake validation mismatch in Agent creation flow

Status Confirmed

Introduced by Version 1

Description The protocol requires users to stake a specific amount of ETH defined in `_agentConfigContract` as `stakeAmount` before invoking the function `createAgentTokenAndPool()` to create relevant `AgentToken` and `AgentPool`. However, this `stakeAmount` is mutable and can be updated between the time a user stakes and when they actually create the `AgentToken` and `AgentPool`. If the value changes during this interval, it may lead to confusion or unexpected reverts, as users might have staked the previously required amount, which becomes insufficient after the change.

```
function requireStakeFor(address staker) public view override returns (uint256) {
    uint256 stakeAmount = _agentConfigContract.getStakeAmount();
    if (staker == address(0)) revert InvalidStaker();
    if (stakes[staker] >= stakeAmount) {
        return 0;
    }
    else {
        return stakeAmount - stakes[staker];
    }
}

/// @inheritdoc IAgentRouter
function stake() external payable override {
    uint256 requireStakeAmount = requireStakeFor(msg.sender);
    if (requireStakeAmount == 0) revert AlreadyStaked();
    if (msg.value != requireStakeAmount) revert InvalidStakeAmount();

    uint256 stakeAmount = msg.value;
    stakes[msg.sender] += stakeAmount;

    emit Staked(msg.sender, stakeAmount, stakes[msg.sender]);
}
```

Listing 2.27: contracts/AgentRouter.sol

```
function createAgentTokenAndPool(
    string memory name,
    string memory symbol,
    uint256 nonce,
    uint256 deadline,
    bytes memory signature
) external returns (address agentToken, address agentPool) {
    uint256 stakeAmount = _agentConfigContract.getStakeAmount();
    if (!stakeReadyFor(msg.sender)) revert InsufficientStake();
    if (address(this).balance < stakeAmount) revert InsufficientBalance();
    stakes[msg.sender] -= stakeAmount;
```

```

        address signer = _agentConfigContract.getAgentSigner();
        if (signer != address(0)) {
            if (block.timestamp > deadline) revert SignatureExpired();

            //
            if (!verifyCreateAgentSignature(msg.sender, name, symbol, nonce, deadline, signature,
                signer)) {
                revert InvalidSignature();
            }
        }

        (agentToken, agentPool) = _factoryContract.createAgentTokenAndPool{value: stakeAmount}(
            msg.sender, // _creator
            name,
            symbol,
            msg.sender, // _initialInvestor
            nonce
        );

        _poolForToken[agentToken] = IAgentPool(agentPool);
        _tokenForPool[agentPool] = IAgentToken(agentToken);
    }
```

Listing 2.28: contracts/AgentRouter.sol

Suggestion Revise the logic accordingly.

Feedback from the project This is an intended behavior. In this sense, users should stake more tokens to create agents.

2.2.6 Redundant code

Status Partially Fixed

Introduced by Version 1

Description There are several unused variables, events, functions. It is recommended to remove them for better code readability. Specifically, the following code should be removed or revised.

In the `_processInitialInvestment()` function, the check for whether `ethIn` is divisible by `INVESTMENT_UNIT` is redundant, as this validation is already handled by `_getAgentTokenOut()`.

The non-zero check for staker in `stakeReadyFor()` is unnecessary, since `requireStakeFor()` performs the same validation.

The `_tokenForPool` variable is written in `createAgentTokenAndPool()` but is never used elsewhere in the protocol, making the assignment redundant.

The event `AddressConfigUpdated` and the error message `TransferFailed` are also defined but not utilized anywhere in the code.

```
function _processInitialInvestment(
    address _initialInvestor,
    uint256 ethIn
```

```
) private returns (uint256 initEthIn, uint256 initAgentTokenOut) {
    if (ethIn % _agentConfigContract.getInvestmentUnit() != 0) revert InvalidInvestAmount();
```

Listing 2.29: contracts/AgentPool.sol

```
function stakeReadyFor(address staker) public view override returns (bool) {
    if (staker == address(0)) revert InvalidStaker();
```

Listing 2.30: contracts/AgentRouter.sol

```
    _tokenForPool[agentPool] = IAgentToken(agentToken);
```

Listing 2.31: contracts/AgentRouter.sol

```
/// @notice
/// @param paramName
/// @param oldValue
/// @param newValue
event AddressConfigUpdated(string paramName, address oldValue, address newValue);
```

Listing 2.32: contracts/interfaces/IAgentConfig.sol

```
/// @notice
error TransferFailed();
```

Listing 2.33: contracts/interfaces/IAgentAirdropTypes.sol

Suggestion Remove the redundant code.

Feedback from the project Partial fixed. Avoid modifying the core contract logic, even if it appears redundant.

Note The event `AddressConfigUpdated` and the error message `TransferFailed` are removed. The other redundant code remains unchanged.

2.3 Note

2.3.1 Signature verification will skip when signer is not set

Introduced by `Version 1`

Description In the current implementation, creating `AgentToken`, `AgentPool`, or making investments into a pool requires a signature from a designated signer. However, if the signer has not been set, the corresponding functions will skip signature verification. The project declares that the signer will be configured during the contract deployment process, and it's allowed to circumvent authorization by setting the signer to the zero address.

```
    if (signer != address(0)) {
        if (block.timestamp > deadline) revert SignatureExpired();

        //
        if (!verifyCreateAgentSignature(msg.sender, name, symbol, nonce, deadline, signature,
            signer)) {
```

```
            revert InvalidSignature();
        }
    }
```

Listing 2.34: contracts/AgentRouter.sol

```
    if (signer != address(0)) {
        if (block.timestamp > deadline) revert SignatureExpired();

        //
        (IAgentPool.Stage stage, , ) = pool.getCurrentStage();

        //
        if (!verifyInvestSignature(msg.sender, agentToken, stage, deadline, signature, signer))
            {
            revert InvalidSignature();
        }
    }
```

Listing 2.35: contracts/AgentRouter.sol

2.3.2 Potential centralization risks

Introduced by `Version 1`

Description In this protocol, several privileged roles (e.g., `Custodian`, `Oracle`) can conduct sensitive operations, which introduces potential centralization risks. For example, an account with `Custodian` privileges can invoke the `withdrawFunds()` function to withdraw native tokens or `ERC20` tokens from the contract. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS