

# Agent

## Smart Contract Security Assessment

Version 1.0

Audit dates: May 28 — May 30, 2024

Audited by: HollaDieWaldfee100  
MiloTruck

# Contents

## 1. Introduction

1.1 About Zenith

1.2 Disclaimer

1.3 Risk Classification

## 2. Executive Summary

2.1 About Agent

2.2 Scope

2.3 Audit Timeline

2.4 Issues Found

## 3. Findings Summary

## 4. Findings

4.1 Low Risk

4.2 Informational

# 1. Introduction

## 1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at <https://code4rena.com/zenith>.

## 1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

## 1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

# 2. Executive Summary

## 2.1 About Agent

Agent provides Web3 users with maximum flexibility and control over their accounts and DeFi earning strategies by enabling the creation and intuitive management of multiple Web3 Sub-Accounts on its Account Platform.

## 2.2 Scope

Repository	<a href="#">AgentExchange/agent-contracts</a>
Commit Hash	<a href="#">ce9e50030fbf28f40326a45aec314077172779c8</a>

## 2.3 Audit Timeline

DATE	EVENT
May 28, 2024	Audit start
May 30, 2024	Audit end
Nov 14, 2024	Report published

## 2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	0
Low Risk	6
Informational	4
Total Issues	10

## 3. Findings Summary

ID	DESCRIPTION	STATUS
L-1	Check length of `feeOracle` signature	Resolved
L-2	ETH cannot be deposited for another user	Resolved

L-3	<code>`AgentPool.deposit()`</code> is incompatible with ERC-777 tokens	Resolved
L-4	Rebasing tokens are not supported	Resolved
L-5	<code>`AgentPool.rescueFunds()`</code> sends funds to the owner or treasury based on <code>`_token`</code>	Resolved
L-6	Missing upper bound check for <code>`fees`</code> in <code>`AgentExchangeV1.initialize()`</code>	Resolved
I-1	Oracle signature does not include <code>`bid.amount`</code> and price based discounts cannot be implemented	Resolved
I-2	<code>`InsufficientBalance`</code> error is unused	Resolved
I-3	Natspec for <code>`AgentExchangeV1.listItem()`</code> has missing parameters	Resolved
I-4	<code>`Initializable`</code> is used without calling <code>`_disableInitializers()`</code> in the constructor	Acknowledged

## 4. Findings

### 4.1 Low Risk

A total of 6 low risk findings were identified.

#### [L-1] Check length of `feeOracle` signature

---

Severity: Low

Status: Resolved

---

Context:

- [AgentExchangeV1.sol#L280-L289](#)

**Description:** For signatures that are provided by bidders, it is checked that their length is equal to 65 bytes. This is to prevent the use of malleable signatures. However, the same check is missing from the `_verifyOracleSignature()` function which checks the signatures of the `feeOracle`.

**Recommendation:** It is recommended to check that the `feeOracle` signature length is equal to 65 bytes.

**Agent:** [Fixed](#).

Zenith: Verified

## [L-2] ETH cannot be deposited for another user

Severity: Low

Status: Resolved

### Context:

- [AgentPool.sol#L51-L66](#)

**Description:** ETH cannot be deposited via `AgentPool.deposit()` since it reverts when `_token=ETH`. It can only be deposited by triggering the `receive()` function.

```
receive() external payable {
    balances[ETH][msg.sender] += msg.value;
    emit Received(ETH, msg.sender, msg.value);
}
```

Since `receive()` always makes the deposit for `msg.sender`, it is not possible to deposit ETH for another user. This makes ETH different from other tokens, which can lead to problems when components integrate with `AgentPool`, expecting that ETH can be deposited for other users.

**Recommendation:** Consider changing `AgentPool.deposit()` to allow ETH deposits.

```
-    function deposit(address _token, address _account, uint256 _amount)
public {
-        if (_token == ETH) {
-            revert InvalidToken();
+    function deposit(address _token, address _account, uint256 _amount)
public payable {
+        if (_token != ETH) {
+            require(msg.value == 0);
        }
        if (_amount == 0) {
            revert InvalidAmount();
        }

-        uint256 balanceBefore = IERC20(_token).balanceOf(address(this));
-        IERC20(_token).safeTransferFrom(msg.sender, address(this),
_amount);
-        uint256 balanceAfter = IERC20(_token).balanceOf(address(this));
+        if (token != ETH) {
+            uint256 balanceBefore =
IERC20(_token).balanceOf(address(this));
```

```

+         IERC20(_token).safeTransferFrom(msg.sender, address(this),
+         _amount);
+         uint256 balanceAfter =
+         IERC20(_token).balanceOf(address(this));

-         balances[_token][_account] += balanceAfter - balanceBefore;
+         balances[_token][_account] += balanceAfter - balanceBefore;
+     } else {
+         require(msg.value == _amount);
+         balances[_token][_account] += _amount;
+     }

    emit Received(_token, _account, _amount);
}

```

Applying this change, it is also possible to remove the `receive()` function.

Agent: [Fixed](#).

**Zenith:** Verified. Also, the `balanceAfter - balanceBefore` difference is no longer checked. Instead it is assumed that no fee-on-transfer tokens will be used and tokens are now whitelisted by the `owner`.



### [L-3] `AgentPool.deposit()` is incompatible with ERC-777 tokens

Severity: Low

Status: Resolved

#### Context:

- [AgentPool.sol#L59-L63](#)

**Description:** Under "ERC20 token behaviors in scope" in the scoping form, "ERC777 used by the protocol" is marked as "Any", which means ERC777 tokens are supported..

When users deposit tokens through `AgentPool1.deposit()`, the amount added to their balance is calculated as the difference between the balance before and after the transfer:

```
uint256 balanceBefore = IERC20(_token).balanceOf(address(this));
IERC20(_token).safeTransferFrom(msg.sender, address(this), _amount);
uint256 balanceAfter = IERC20(_token).balanceOf(address(this));

balances[_token][_account] += balanceAfter - balanceBefore;
```

However, when `_token` is an ERC777 token, the caller can use the `tokensToSend` hook to re-enter `deposit()` to deposit more tokens during the transfer. For example:

- Call `deposit()` with 100 tokens.
  - Assume `balanceBefore = 0`.
  - `_token.transferFrom()` is called, which calls the `tokensToSend` hook and gives execution control to `msg.sender`:
    - The caller re-enters `deposit()` and deposits another 100 tokens. This adds 100 tokens to the caller's balance.
  - `_token.transferFrom()` transfers 100 tokens to the contract.
  - `balanceAfter = 200`, which adds 200 tokens to the caller's balance.
- The caller now has a balance of 300 tokens, even though he only deposited 200 tokens.

As seen from above, `balanceAfter - balanceBefore` in the initial `deposit()` call will also include the balance of the re-entered `deposit()` call, causing a double-counting of the user's deposit.

**Recommendation:** Consider reverting the change in `Agent.deposit()` to not support fee-on-transfer tokens:

```
balances[_token][_account] += _amount;
```

```
IERC20(_token).safeTransferFrom(msg.sender, address(this), _amount);
```

Otherwise, document that ERC777 tokens are not supported.

Agent: [Fixed](#).

**Zenith:** Verified. Tokens that are not whitelisted by the `owner` cannot be deposited into `AgentPool` and listings with non-whitelisted tokens cannot be created.

## [L-4] Rebasing tokens are not supported

Severity: Low

Status: Resolved

### Context:

- [AgentPool.sol#L59-L63](#)

**Description:** Under "ERC20 token behaviors in scope" in the scoping form, tokens with "[balance changes outside of transfers](#)" (ie. rebasing tokens) are marked as supported.

However, the current design of `AgentPool` is not able to support rebasing tokens, especially ones where balances can decrease. When users deposit tokens through `AgentPool.deposit()`, the amount deposited is recorded in a `balances` mapping:

```
uint256 balanceBefore = IERC20(_token).balanceOf(address(this));
IERC20(_token).safeTransferFrom(msg.sender, address(this), _amount);
uint256 balanceAfter = IERC20(_token).balanceOf(address(this));

balances[_token][_account] += balanceAfter - balanceBefore;
```

These token balances are then used to trade NFTs in `AgentExchangeV1`.

However, if `_token` is a rebasing token, over time, the `balances` mapping will not accurately reflect the amount of tokens belonging to a user. For example, if `_token` rebases down, the actual token balance held in the contract will be lower than the amount stored in the `balances` mapping.

This could cause unexpected behavior in the protocol, for example, `AgentPool.withdraw()` could revert even when the user's `balances` mapping is more than `_amount`.

Note that on Blast L2, WETH and USDB are rebasing tokens by default. However, they can be used in the protocol as both tokens only rebase up (ie. the balance held by an address never decreases). They can also be configured to not rebase by calling `IERC20Rebasing.configure()` for both tokens in `AgentPool.initialize()`.

### Recommendation:

Consider adding a token whitelist to `AgentPool`, which prevents rebasing tokens and other incompatible tokens from being used.

Alternatively, document that `AgentPool` is not compatible with rebasing tokens.

Agent: [Fixed](#).

**Zenith:** Veified. The issue has been fixed by implementing a whitelist. Tokens that are not whitelisted by the `owner` cannot be deposited into `AgentPool` and listings with non-whitelisted tokens cannot be created.

[L-5] `AgentPool.rescueFunds()` sends funds to the owner or treasury based on `\_token`

Severity: Low

Status: Resolved

Context:

- [AgentPool.sol#L126-L131](#)

**Description:** In `AgentPool.rescueFunds()`, ETH is rescued to `msg.sender` whereas other tokens are rescued to `treasury`:

```
if (_token == ETH) {
    (bool sent,) = msg.sender.call{value: _amount}("");
    require(sent, "Failed to send Ether");
} else {
    IERC20(_token).safeTransfer(treasury, _amount);
}
```

This seems inconsistent - tokens should always be rescued to one address, regardless of when it is ETH or other ERC20 tokens.

**Recommendation:** Amend `rescueFunds()` to either ETH to the `treasury`, or send all other tokens to `msg.sender`.

**Agent:** [Fixed](#).

**Zenith:** Verified. All rescued funds are sent to the `treasury`.

## [L-6] Missing upper bound check for `fees` in `AgentExchangeV1.initialize()`

Severity: Low

Status: Resolved

### Context:

- [AgentExchangeV1.sol#L248-L254](#)
- [AgentExchangeV1.sol#L67](#)

**Description:** In `AgentExchangeV1.sol`, when fees are set by the owner using `setFees()`, they are checked to be less than or equal to `1e8`:

```
function setFees(uint256 _fees) external onlyOwner {
    if (_fees > 1e8) {
        revert InvalidFees();
    }
    fees = _fees;
```

However, this check does not exist in `initialize()` even though `fees` is also set. As a result, when the `AgentExchangeV1` contract is first initialized, `fees` can be set to any arbitrary value greater than `1e8`.

**Recommendation:** In `AgentExchangeV1.initialize()`, check that `_fees` is not greater than `1e8`:

```
function initialize(...)
    public
    initializer
{
+   if (_fees > 1e8) revert InvalidFees();
    _initializeOwner(_owner);
    pool = IAgentPool(_pool);
    feeOracle = _feeOracle;
    fees = _fees;
```

Agent: [Fixed](#).

Zenith: Verified

## 4.2 Informational

A total of 4 informational findings were identified.

### [I-1] Oracle signature does not include `bid.amount` and price based discounts cannot be implemented

---

Severity: Informational

Status: Resolved

---

#### Context:

- [AgentExchangeV1.sol#L186-L219](#)

#### Description:

In `AgentExchangeV1.takeBid()`, it is possible that `item.amount != bid.amount`. This means buyer and seller can agree on a price that is different from the price of the listing.

However, the oracle only signs the price in `item.amount`, not `bid.amount`, so the oracle cannot consider the actual price of the sale when calculating its discount. It has been determined by the client that this is not a concern since the `discount` will not depend on the price of the sale.

**Recommendation:** To allow for discount models that rely on the price of the sale, it is recommended to include `bid.amount` in the data that the `feeOracle` needs to sign.

**Agent:** Fixed [here](#) and [here](#).

**Zenith:** Verified. For bids, `bid.amount` is part of the oracle signature. For asks, `item.amount` is part of the oracle signature.

## [I-2] `InsufficientBalance` error is unused

---

Severity: Informational

Status: Resolved

---

Context: [IAgentExchangeV1Utils.sol#L31](#)

**Description/Recommendation:** Consider removing the `InsufficientBalance` error from `IAgentExchangeV1Utils` since it is never used anywhere.

Agent: [Fixed](#).

Zenith: Verified.



### [I-3] Natspec for `AgentExchangeV1.listItem()` has missing parameters

Severity: Informational

Status: Resolved

#### Context:

- [AgentExchangeV1.sol#L71-L77](#)

#### Description/Recommendation:

The natspec for `AgentExchangeV1.listItem()` is missing the `token` and `expiry` parameter:

```
/*
 * @notice Method for listing NFT
 * @param nftAddress Address of NFT contract
 * @param tokenId Token ID of NFT
+ * @param token sale token for each item
 * @param price sale price for each item
+ * @param expiry expiry timestamp for the listing
 */
function listItem(address nftAddress, uint256 tokenId, address token,
uint256 price, uint256 expiry)
```

Agent: [Fixed](#).

Zenith: Verified.

#### [I-4] `Initializable` is used without calling `\_disableInitializers()` in the constructor

Severity: Informational

Status: Acknowledged

##### Context:

- [AgentPool.sol](#)
- [AgentExchangeV1.sol](#)

##### Description:

[Solady's Initializable.sol](#) recommends calling `_disableInitializers()` in the constructor of implementation contracts:

```
/// @dev Locks any future initializations by setting the initialized
version to `2**64 - 1`.
///
/// Calling this in the constructor will prevent the contract from being
initialized
/// or reinitialized. It is recommended to use this to lock
implementation contracts
/// that are designed to be called through proxies.
///
/// Emits an {Initialized} event the first time it is successfully
called.
function _disableInitializers() internal virtual {
```

This prevents users from initializing the implementation contract, which could affect the proxy contract under certain conditions (eg. the implementation contract can be self-destructed).

In this protocol, an attacker initializing the implementation contracts of `AgentPool` or `AgentExchangeV1` will have no impact on the proxy contracts. Nevertheless, it is best practice to call `_disableInitializers()` to prevent attackers from doing so.

**Recommendation:** In both `AgentPool` and `AgentExchangeV1`, add a constructor that calls `_disableInitializers()`:

```
constructor() {
    _disableInitializers();
}
```

**Agent:** Acknowledged. It doesn't affect proxied contracts, also our implementation will not be self destructable so we can skip this.