RENASCENCE

Agent Audit Report

Version 2.0

Audited by:

MiloTruck

HollaDieWaldfee

June 13, 2024

Contents

1 Introduction

1.1 About Renascence

Renascence Labs was established by a team of experts including HollaDieWaldfee, MiloTruck, alexxander and bytes032.

Our founders have a distinguished history of achieving top honors in competitive audit contests, enhancing the security of leading protocols such as Reserve Protocol, Arbitrum, MaiaDAO, Chainlink, Dodo, Lens Protocol, Wenwin, PartyDAO, Lukso, Perennial Finance, Mute and Taurus.

We strive to deliver tailored solutions by thoroughly understanding each client's unique challenges and requirements. Our approach goes beyond addressing immediate security concerns; we are dedicated to fostering the enduring success and growth of our partners.

More of our work can be found here.

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

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	High	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

1.3.1 Impact

- High - Funds are **directly** at risk, or a **severe** disruption of the protocol's core functionality
- Medium - Funds are **indirectly** at risk, or **some** disruption of the protocol's functionality
- Low - Funds are **not** at risk

1.3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

2 Executive Summary

2.1 About Agent

Agent is a platform for creating, managing, and trading Web3 Sub-Accounts. It is powered by the ERC-6551 token standard and is advised by ERC-6551 co-author Vectorized. Agent is live on Ethereum mainnet and can support any EVM-compatible chain in the future. Agent aims to accelerate the adoption of smart contract accounts ("acc/acc"). Agent is not competitive with, but rather complimentary to, other efforts like ERC-4337.

2.2 Overview

Project	Agent
Repository	agent-contracts
Commit Hash	af47543c255a...
Mitigation Hash	9d2bda69f352...
Date	28 May 2024 - 29 May 2024

2.3 Issues Found

Severity	**Count**
High Risk	0
Medium Risk	0
Low Risk	6
Informational	4
Total Issues	**10**

3 Findings Summary

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

4 Findings

Low Risk

[L-1] Missing upper bound check for `fees` in `AgentExchangeV1.initialize()`

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.

Renascence: The recommendation has been implemented.

[L-2] `AgentPool.rescueFunds()` sends funds to the owner or treasury based on `_token`

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.

Renascence: The recommendation has been implemented. All rescued funds are sent to the `treasury`.

[L-3] Rebasing tokens are not supported

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.

Renascence: 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-4] `AgentPool.deposit()` is incompatible with ERC-777 tokens

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 `AgentPool.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.

Renascence: 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] ETH cannot be deposited for another user

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.

Renascence: The recommendation has been implemented. 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-6] Check length of `feeOracle` signature

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.

Renascence: The recommendation has been implemented.

Informational

[I-1] `Initializable` is used without calling `_disableInitializers()` in the constructor

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.

Renascence: The finding does not have a security impact. It is okay to acknowledge it.

[I-2] Natspec for `AgentExchangeV1.listItem()` has missing parameters

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.

Renascence: The recommendation has been implemented.

[I-3] `InsufficientBalance` error is unused

Context: IAgentExchangeV1Utils.sol#L31

Description/Recommendation:

Consider removing the `InsufficientBalance` error from `IAgentExchangeV1Utils` since it is never used anywhere.

Agent: Fixed.

Renascence: The recommendation has been implemented.

[I-4] Oracle signature does not include `bid.amount` and price based discounts cannot be implemented

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.

Renascence: Fixed as recommended. For bids, `bid.amount` is part of the oracle signature. For asks, `item.amount` is part of the oracle signature.

5 Centralization risks

5.1 Proxy owner is fully trusted

Both AgentPool and AgentExchangeV1 are deployed behind proxies, and according to the client, the proxy owner is a multisig that is separate from the owner role within the contracts. The proxy owner can upgrade both contracts to an arbitrary implementation, thereby it is fully trusted.

5.2 Owner is fully trusted

The owner of AgentPool and AgentExchangeV1 has full control over the tokens and ETH in AgentPool by either using AgentPool.rescueFunds() or by setting a different exchange which can then call AgentPool.transferFrom(). In AgentExchangeV1, the owner can gain access to all NFTs indirectly. Since the owner has access to all funds in AgentPool, it can just set an oracle and purchase all NFTs with AgentExchangeV1.takeAsk(), then get the paid funds back.

5.3 Oracle can DoS buyers and sellers

By not providing its service, the oracle can DoS the AgentExchangeV1 contract since both takeAsk() and takeBid() require valid oracle signatures. A malicious oracle can also tamper with the dicount rate which can have small effects on sellers' proceeds. The owner can replace a misbehaving oracle by calling setFeeOracle().

6 Systemic risks

6.1 External tokens risk

In contrast to NFTs in AgentExchangeV1 which are whitelisted, AgentPool can interact with any ERC20 tokens and NFTs can be listed to be sold for any token. Each token must be assessed individually based on its centralization and systemic risks. Users must ensure they only interact with tokens they trust.

6.2 Buyers and sellers can front-run each other

NFT listings can be cancelled at any time by the seller. And both buyers and sellers can transfer their funds in AgentPool. Thereby it is possible that buyer and seller agree on a sale, but then pull out of the agreement by front-running each other. A sale can only be considered closed after the transaction has been executed. In particular, the existence of valid signatures is not a guarantee that the sale can take place.