Scanned Code Report

AUDITAGENT

Powered by NETHERMIND SECURITY

Code Info

Deep Scan

Scan ID 1	Date August 29, 2025
Organization ChaosChain	Repository trustless-agents-erc-ri
Branch main	Commit Hash 740ef781...0141f4ea

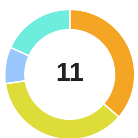
Contracts in scope

src/IdentityRegistry.sol src/ReputationRegistry.sol src/ValidationRegistry.sol

Code Statistics

Findings 11	Contracts Scanned 3	Lines of Code 488

Findings Summary



Total Findings

- High Risk (0)
- Medium Risk (4)
- Low Risk (4)
- Info (1)
- Best Practices (2)

Code Summary

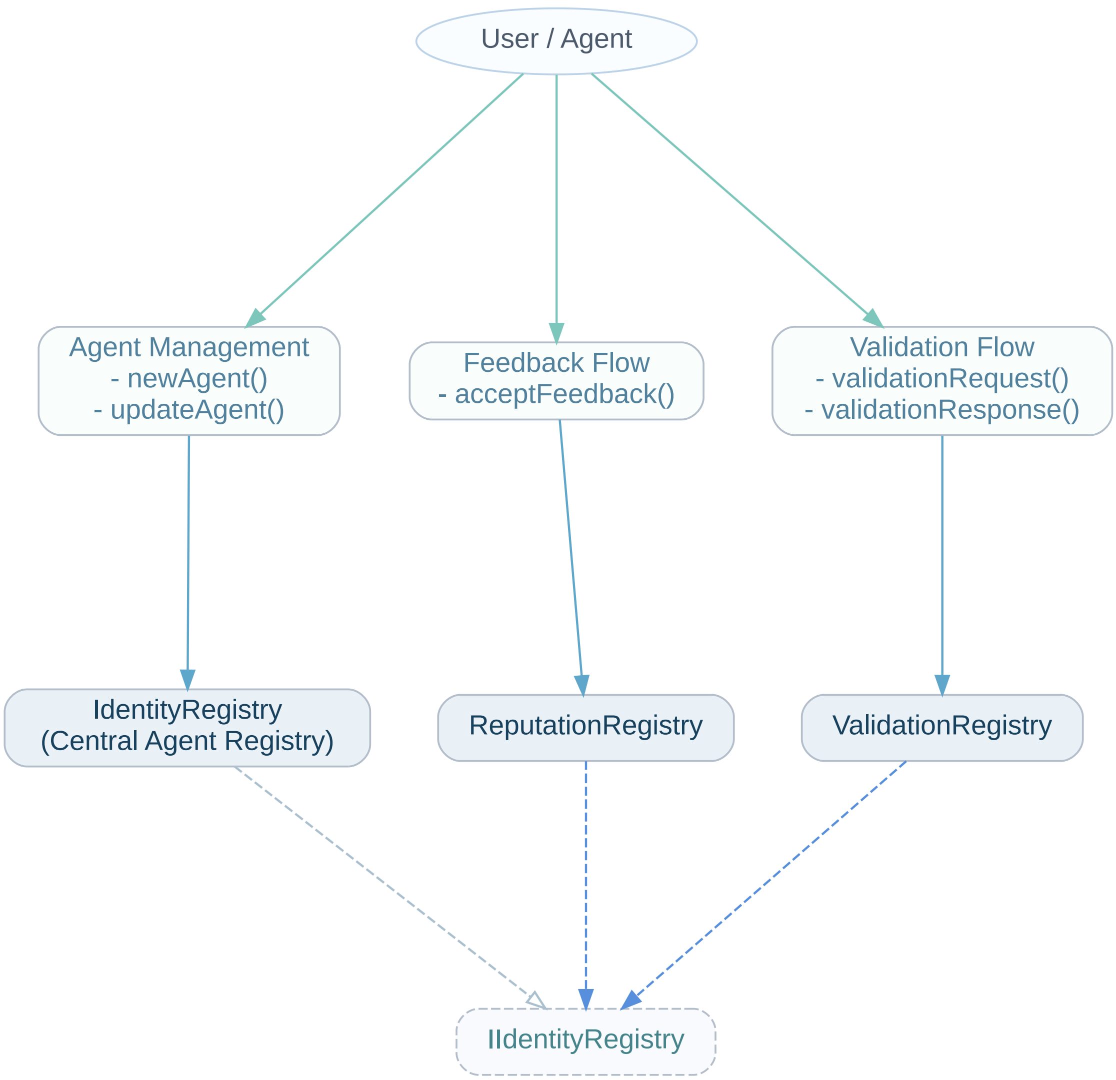
The protocol establishes a comprehensive framework for "Trustless Agents" by providing decentralized identity, reputation, and validation services through a suite of interconnected smart contracts. It is designed to create a verifiable and trustworthy ecosystem for autonomous agents or users to interact.

The system is composed of three core components:
- **IdentityRegistry**: This is the foundational contract that serves as a central registry for all agent identities. Any user can register a new agent by providing a unique domain name and an associated Ethereum address, and paying a small, one-time fee that is burned to prevent spam. Each registered agent receives a unique numerical ID. Agent owners can later update their registered domain or address.
- **ReputationRegistry**: This contract builds upon the identity system to manage feedback between agents. It enables a "server" agent to explicitly authorize a "client" agent to provide feedback for a specific interaction. This creates a verifiable, on-chain authorization record, forming the basis for a lightweight reputation system.
- **ValidationRegistry**: This contract facilitates a process for independent, third-party validation of data or tasks. Any user can submit a validation request for a specific piece of data (represented by a hash), designating a "server" agent and a "validator" agent. The designated validator agent is then authorized to submit a quantitative response (a score from 0 to 100) within a specific time window, which is permanently recorded on-chain.

Main Entry Points and Actors

- `newAgent(string agentDomain, address agentAddress)`: Allows **anyone** to register a new agent identity by paying a registration fee.
- `updateAgent(uint256 agentId, string newAgentDomain, address newAgentAddress)`: Allows an **Agent Owner** to update the domain or address associated with their agent ID.
- `acceptFeedback(uint256 agentClientId, uint256 agentServerId)`: Allows a **Server Agent Owner** to authorize a client agent to provide feedback.
- `validationRequest(uint256 agentValidatorId, uint256 agentServerId, bytes32 dataHash)`: Allows **anyone** to request validation for a piece of data from a designated validator agent.
- `validationResponse(bytes32 dataHash, uint8 response)`: Allows a designated **Validator Agent Owner** to submit a response to a pending validation request.

Code Diagram



1 of 11 Findings

src/ValidationRegistry.sol

State / event desynchronisation in ValidationRegistry.validationRequest(...)

● Medium Risk

If a validation request already exists for a given `dataHash` and is still within the `EXPIRATION_SLOTS` window, calling `validationRequest(...)` again does **not** revert. Instead, the function emits a fresh `ValidationRequestEvent` that contains the *new* `agentValidatorId` and `agentServerId` supplied by the caller but exits **before** mutating storage.

```
if (existingRequest.dataHash != bytes32(0)) {
    if (block.number <= existingRequest.timestamp + EXPIRATION_SLOTS) {
        // <== Storage is **not** updated
        emit ValidationRequestEvent(agentValidatorId, agentServerId, dataHash);
        return;                     // <== early exit
    }
}
```

Consequences:
1. Off-chain indexers that rely on the event stream will believe that a new validator/server pair is authorised, while on-chain state still points to the original validator/server. The “new” validator will later revert when it tries to answer because `msg.sender` will not match the stored validator address.
2. A malicious actor can grief other servers by front-running their request with the same `dataHash`, forcing them to waste gas on an apparently successful transaction that will never be honourable.
3. The invariant *“validationRequest cannot overwrite an unexpired existing request”* is silently violated at the event level.

The bug directly undermines the correctness of the validation flow and can be exploited permission-lessly; hence the high severity.

2 of 11 Findings

src/IdentityRegistry.sol

Case / Unicode differences bypass the "unique domain" invariant

● Medium Risk

Domain uniqueness is enforced with

```
if (_domainToAgentId[agentDomain] != 0) revert DomainAlreadyRegistered();
```

The key of the mapping is the raw `string` bytes. Because DNS names are defined to be **case-insensitive** and susceptible to Unicode normalisation attacks, an attacker can register visually identical or equivalent domains, e.g. `"Example.com"`, `"example.com"`, `"ｅxample.com"` (full-width 'e'), etc. All these byte-wise different strings will pass the uniqueness check even though they represent the same domain to human users, breaking the intended one-to-one mapping between real-world domains and agents.

3 of 11 Findings

src/IdentityRegistry.sol

Domain registration is vulnerable to front-running

Medium Risk

The newAgent function performs the uniqueness check for agentDomain only when the transaction is mined. Observers can front-run a pending registration by submitting their own transaction with the same domain and higher priority, thereby claiming the domain first and forcing the legitimate user's transaction to revert, resulting in gas loss and permanent domain hijacking.

4 of 11 Findings

src/ValidationRegistry.sol

Risky Strict Equality Check

Medium Risk

The `ValidationRegistry` contract uses a dangerous strict equality check when comparing a hash value to zero:

```
request.dataHash == bytes32(0)
```

This check occurs in the `getValidationRequest` function at line 138. Using strict equality (`==`) with bytes32 values can be problematic when validating if a value is empty or uninitialized.

Strict equality checks can lead to false negatives in validation logic, as there might be edge cases where a value is not exactly zero but should still be considered invalid. This could potentially allow manipulation of the validation request system.

Consider using a more robust validation approach or explicitly documenting why this strict equality check is safe in this specific context.

Self-Validation Vulnerability in ValidationRegistry

Low Risk

The `validationRequest` function in the ValidationRegistry contract allows an agent to request validation from any other agent, including themselves. This creates a potential conflict of interest where an agent could validate their own data.

```
function validationRequest(
    uint256 agentValidatorId,
    uint256 agentServerId,
    bytes32 dataHash
) external {
    // Validate inputs
    if (dataHash == bytes32(0)) {
        revert InvalidDataHash();
    }

    // Validate that both agents exist
    if (!identityRegistry.agentExists(agentValidatorId)) {
        revert AgentNotFound();
    }
    if (!identityRegistry.agentExists(agentServerId)) {
        revert AgentNotFound();
    }

    // Check if request already exists and is still valid
    IValidationRegistry.Request storage existingRequest = _validationRequests[dataHash];
    if (existingRequest.dataHash != bytes32(0)) {
        if (block.number <= existingRequest.timestamp + EXPIRATION_SLOTS) {
            // Request still exists and is valid, just emit the event again
            emit ValidationRequestEvent(agentValidatorId, agentServerId, dataHash);
            return;
        }
    }

    // Create new validation request
    _validationRequests[dataHash] = IValidationRegistry.Request({
        agentValidatorId: agentValidatorId,
        agentServerId: agentServerId,
        dataHash: dataHash,
        timestamp: block.number,
        responded: false
    });

    emit ValidationRequestEvent(agentValidatorId, agentServerId, dataHash);
}
```

Notice that there is no check to prevent `agentValidatorId` from being the same as `agentServerId`. This means an agent could create a validation request where they are both the server and the validator, and then provide their own validation response. This undermines the integrity of the validation system, as the purpose of validation is to have independent verification of data.

A malicious agent could exploit this to create fake validations that appear legitimate, potentially misleading other users of the system who rely on these validations for decision-making.

6 of 11 Findings src/ValidationRegistry.sol

Stale validation responses persist when a new request overwrites an old one • Low Risk

When validationRequest is called with a dataHash that previously had a request which has now expired, the contract overwrites `_validationRequests[dataHash]` **without** clearing `_validationResponses[dataHash]` and `_hasResponse[dataHash]`. A fresh request therefore starts with `request.responded == false` while `getValidationResponse` still returns `hasResponse == true` and an outdated score.

```
// ValidationRegistry.sol
_validationRequests[dataHash] = IValidationRegistry.Request({
    agentValidatorId: agentValidatorId,
    agentServerId: agentServerId,
    dataHash: dataHash,
    timestamp: block.number,
    responded: false
});
// no: delete _validationResponses[dataHash];
// no: delete _hasResponse[dataHash];
```

Attackers can recycle favourable old scores or mislead off-chain reputation systems that only look at the existence of a response, breaking the one-request-one-response invariant.

src/IdentityRegistry.sol

Flawed Uniqueness Check in `updateAgent` Prevents Legitimate Updates

● Low Risk

The `updateAgent` function in the `IdentityRegistry` contract prevents an agent owner from updating their agent's information if they provide their current domain or address as one of the parameters. The function checks if a new domain or address is already registered by querying the `_domainToAgentId` and `_addressToAgentId` mappings. However, the check does not account for the case where the domain or address belongs to the very agent being updated. For example, if an agent owner tries to update only their `agentAddress`, they might intuitively pass their existing `agentDomain` along with the new address. The function will incorrectly detect that the domain is already registered (to themselves) and revert the transaction. This forces the agent owner to use a non-obvious method for partial updates (i.e., passing an empty string for the domain or a zero address for the address they do not wish to change), creating a significant usability issue and a potential footgun that blocks valid state changes.

```
// File: src/IdentityRegistry.sol

function updateAgent(
    uint256 agentId,
    string calldata newAgentDomain,
    address newAgentAddress
) external returns (bool success) {
    // ...
    bool domainChanged = bytes(newAgentDomain).length > 0;
    // ...
    if (domainChanged) {
        if (_domainToAgentId[newAgentDomain] != 0) { // This check fails if newAgentDomain
is the agent's current domain
            revert DomainAlreadyRegistered();
        }
    }
    // ...
}
```

8 of 11 Findings src/IdentityRegistry.sol src/ReputationRegistry.sol src/ValidationRegistry.sol

PUSH0 Opcode Compatibility Issue

Low Risk

The contracts use pragma `^0.8.19`, which means they could potentially be compiled with Solidity 0.8.20 or later. Starting from version 0.8.20, the Solidity compiler defaults to targeting the Shanghai EVM version, which introduces the PUSH0 opcode.

This presents a compatibility risk when deploying to chains that haven't implemented the Shanghai upgrade or L2 solutions that may not support the PUSH0 opcode. On these chains, contract deployment will fail if the bytecode contains PUSH0 instructions.

Affected chains might include various Layer 2 solutions, sidechains, or EVM-compatible blockchains that haven't implemented the Shanghai upgrade.

To mitigate this issue:
1. Either lock the pragma to a specific version before 0.8.20 (e.g., `pragma solidity 0.8.19;`)
2. Or explicitly set the EVM version target in your compiler settings if using 0.8.20+ (e.g., using the `--evm-version` flag with `solc` or configuring it in your build tool)

9 of 11 Findings src/ValidationRegistry.sol

Unbounded storage growth – expired or completed validation requests are never deleted

Info

Both `_validationRequests` and `_validationResponses` mappings grow for every `validationRequest` ever made. Neither `validationResponse` nor any maintenance routine deletes the entry once it has expired (after 1 000 blocks) or after a response is recorded. Over time, especially on high-traffic deployments, this leads to an ever-increasing storage footprint, raising the gas-cost of `SLOAD`/`SSTORE` operations and creating long-term state-bloat risks for the network.

10 of 11 Findings

src/ValidationRegistry.sol

Redundant State for Tracking Validation Responses

• Best Practices

The `ValidationRegistry` contract uses two separate state variables to track whether a validation request has been answered: the `responded` boolean within the `Request` struct (stored in the `_validationRequests` mapping) and a separate `_hasResponse` boolean mapping. Both variables are set to `true` within the `validationResponse` function to indicate the same state.

```
// State variables
mapping(bytes32 => IValidationRegistry.Request) private _validationRequests;
// ...
mapping(bytes32 => bool) private _hasResponse; // <-- Redundant

// in validationResponse function
// ...
        // Mark as responded and store the response
        request.responded = true; // <-- First flag
        _validationResponses[dataHash] = response;
        _hasResponse[dataHash] = true; // <-- Second, redundant flag
// ...
```

This redundancy increases gas costs for storage writes and adds unnecessary complexity. The `_hasResponse` mapping could be removed, and the `getValidationResponse` function could be modified to use the `responded` flag from the `_validationRequests` mapping, simplifying the contract and saving gas.

11 of 11 Findings

src/IdentityRegistry.sol

Updating Agent with No Changes Wastes Gas and Emits Misleading Event

• Best Practices

The `updateAgent` function can be successfully called with a `newAgentDomain` that is an empty string and a `newAgentAddress` that is the zero address. In this scenario, the function performs no state changes but still proceeds to the end, emits an `AgentUpdated` event with the old data, and returns `true`. This behavior is inefficient, as it consumes gas for a transaction that has no effect, and can be confusing for off-chain services that monitor events, as an `AgentUpdated` event is emitted without any actual update occurring.

Disclaimer

Kindly note, the Audit Agent is currently in beta stage and no guarantee is being given as to the accuracy and/or completeness of any of the outputs the Audit Agent may generate, including without limitation this Report. The results set out in this Report may not be complete nor inclusive of all vulnerabilities. The Audit Agent is provided on an 'as is' basis, without warranties or conditions of any kind, either express or implied, including without limitation as to the outputs of the code scan and the security of any smart contract verified using the Audit Agent.

Blockchain technology remains under development and is subject to unknown risks and flaws. This Report does not indicate the endorsement of any particular project or team, nor guarantee its security. Neither you nor any third party should rely on this Report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset.

To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this Report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate.

FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.