Scanned Code Report

AUDITAGENT

Powered by NETHERMIND SECURITY

Code Info

Deep Scan

Scan ID 3	Date August 31, 2025
Organization ChaosChain	Repository trustless-agents-erc-ri
Branch main	Commit Hash 0266364d...9d9d9593

Contracts in scope

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

Code Statistics

Findings 11	Contracts Scanned 3	Lines of Code 530

Findings Summary

Total Findings

- High Risk (1)
- Medium Risk (2)
- Low Risk (4)
- Info (3)
- Best Practices (1)

Code Summary

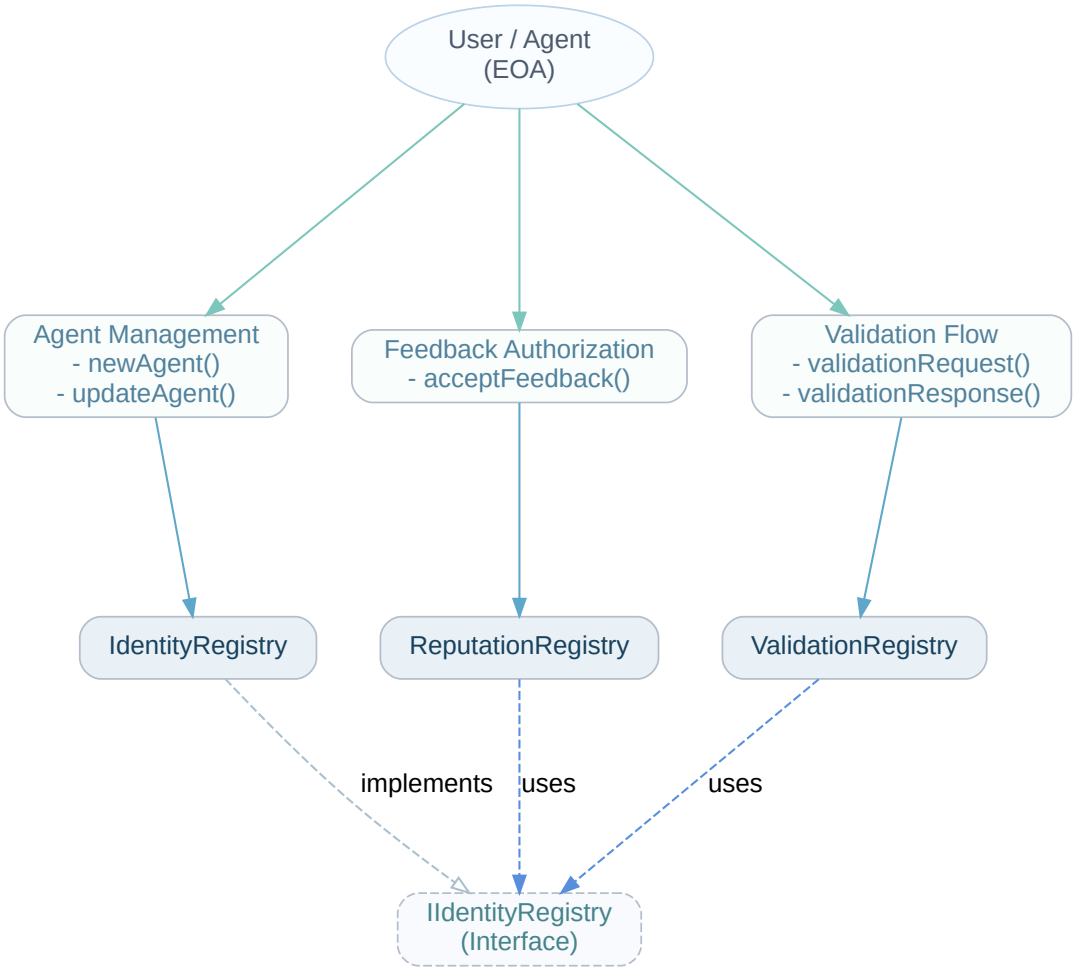
This protocol establishes a decentralized framework for identity, reputation, and validation for entities referred to as "Trustless Agents." It is composed of three core, interconnected registries designed to build trust and accountability in a decentralized environment.

The architecture is modular:
- **IdentityRegistry**: This contract serves as the foundational layer, providing a central registry for all agent identities. Agents can register a unique identity by linking a human-readable domain name to their on-chain address. This creates a verifiable, on-chain record that prevents impersonation and serves as a single source of truth for agent identification.
- **ReputationRegistry**: Building upon the identity layer, this contract manages feedback authorizations between agents. It does not store reputation scores directly but instead provides a lightweight mechanism for a "server" agent to grant on-chain permission for a "client" agent to provide feedback. This creates a verifiable authorization record that can be used by off-chain reputation systems to ensure the legitimacy of feedback.
- **ValidationRegistry**: This contract provides a system for independent, on-chain validation of off-chain data or tasks. An agent can request that a specific piece of data (represented by a data hash) be validated by a designated "validator" agent. The validator can then submit a response, typically a score, which is recorded on-chain. This creates a transparent and immutable record of third-party verification.

Main Entry Points and Actors

- `newAgent(string, address)`: An **Agent** registers a new identity, linking a domain to their address.
- `updateAgent(uint256, string, address)`: An **Agent** updates their own registered domain or address information.
- `acceptFeedback(uint256, uint256)`: A **Server Agent** authorizes a **Client Agent** to provide feedback, creating an on-chain authorization record.
- `validationRequest(uint256, uint256, bytes32)`: Any **Agent** can request validation for a specific data hash from a designated **Validator Agent** on behalf of a **Server Agent**.
- `validationResponse(bytes32, uint8)`: A designated **Validator Agent** submits a validation score for a pending request, which is then recorded on-chain.

Code Diagram



1 of 11 Findings

src/IdentityRegistry.sol

Domain-normalization bug in `updateAgent` corrupts the forward / reverse mappings

High Risk

`newAgent(..)` lower-cases the domain before it is used as a key in `_domainToAgentId`, while the **original** (mixed-case) string is kept only in `_agents[agentId].agentDomain` for display purposes:

```
string memory normalized = _toLowercase(agentDomain);
_domainToAgentId[normalized] = agentId;   // stored with lower-case key
_agents[agentId].agentDomain = agentDomain; // mixed-case copy only
```

`updateAgent(..)` tries to remove the old entry and insert the new one, **but it never normalises either string**:

```
// 1. duplicate check uses the raw string — bypassed with case-variance
if (_domainToAgentId[newAgentDomain] != 0) revert DomainAlreadyRegistered();

// 2. delete uses the mixed-case string stored in `_agents`
delete _domainToAgentId[agent.agentDomain];          // <- NO match – slot is never cleared

// 3. insertion again stores the raw user input
_domainToAgentId[newAgentDomain] = agentId;          // <- stored with mixed case
```

Consequences:
1. **Stale entry** – the lower-cased old domain is *not* deleted, so `_domainToAgentId[oldDomain]` still resolves to the same `agentId` while the agent now points to another domain. Two different domains now map to the same ID, breaking the bidirectional-mapping invariant.
2. **Broken resolution** – the newly inserted key is *mixed-case*, but `resolveByDomain(..)` lower-cases the user-supplied string before performing the lookup. Every call will therefore miss the mixed-case key and revert with `AgentNotFound` → the agent becomes un-resolvable by its own (updated) domain.
3. **Duplicate bypass** – because the duplicate check also uses the mixed-case string, it can be evaded by registering `"Example.com"` after `"example.com"` (or any other case variation), violating the uniqueness guarantees and enabling impersonation attacks.

The bug directly violates three stated invariants:
* “Bidirectional mappings stay consistent after any write”
* “Domains are unique (case-insensitive)”
* “Resolver functions always return an agent whose ID matches the mapping key used”

Any user can corrupt the registry and make his own identity unreachable by simply calling `updateAgent` once with a different letter-case.

2 of 11 Findings

src/IdentityRegistry.sol

Missing registration-fee check enables unlimited identity spam and state bloat • Medium Risk

The documentation specifies a 0.005 ETH fee that *“is burned / locked to prevent spam”*. `newAgent(..)` does **not** verify `msg.value`, therefore anybody can register an unlimited number of agents at zero cost:

```
function newAgent(string calldata agentDomain, address agentAddress) external returns
(uint256) {
    // <- no `require(msg.value == 0.005 ether)` or similar
```

Because every successful registration stores:
* one `AgentInfo` struct,
* one domain-to-id entry,
* one address-to-id entry,

an attacker can continuously create identities and inflate contract-state indefinitely. Apart from the permanent storage rent paid by the protocol, many read-only functions become increasingly more expensive (and might eventually run out of gas in off-chain calls executed via `eth_call`).

Example grief:

```
for (uint i; i < 1_000_000; ++i) {
    identity.newAgent{value:0}(string.concat("a",Strings.toString(i)),
address(uint160(i+1)));
}
```

After ≈1 M registrations the contract would consume roughly 70 MB of state, and every `resolveByDomain` / `newAgent` call incurs significant additional gas. The intended spam-prevention mechanism is therefore ineffective.

3 of 11 Findings

src/ValidationRegistry.sol

Unbounded growth of `_validationRequests` allows permanent storage-bloat attacks

Medium Risk

`validationRequest(...)` stores every unique `dataHash` inside `_validationRequests` and **never deletes the entry**, even after the request has expired or has been answered:

```
if (existingRequest.dataHash != 0) {
    if (expired) {
        delete _validationResponses[dataHash]; // response cleared
        // request struct is *kept* and will immediately be overwritten below
    }
}
_validationRequests[dataHash] = Request({ ... });
```

An attacker can therefore keep submitting requests with fresh random `dataHash` values (or cycle through expired ones that are overwritten) at a cost of only ~20 k gas and **0 ETH** each. Every call appends a new `Request` struct (32 bytes * 5 fields = 160 bytes) to contract storage, and nothing ever reclaims it.

After a large number of calls, the contract's state size and therefore the gas cost of *all* subsequent write operations increase. In extreme cases, benign users could be priced out of the network because their transactions surpass the block-gas-limit.

A minimal PoC that clogs the registry:

```
bytes32 h = keccak256(abi.encodePacked(block.timestamp, msg.sender));
validation.validationRequest(validatorId, serverId, h);
// repeat in a loop or across many txs
```

Mitigations announced in the specification ("time-bounded requests are *automatically cleaned up*") are absent in the implementation.

4 of 11 Findings

src/IdentityRegistry.sol

Potential Front-Running in Agent Registration

Low Risk

The `newAgent` function in the IdentityRegistry contract is susceptible to front-running attacks where an attacker could monitor the mempool for agent registration transactions and submit their own transaction with a higher gas price to register the same domain name first.

```
function newAgent(
    string calldata agentDomain,
    address agentAddress
) external returns (uint256 agentId) {
    // SECURITY: Only allow registration of own address to prevent impersonation
    if (msg.sender != agentAddress) {
        revert UnauthorizedRegistration();
    }

    // Validate inputs
    if (bytes(agentDomain).length == 0) {
        revert InvalidDomain();
    }
    if (agentAddress == address(0)) {
        revert InvalidAddress();
    }

    // SECURITY: Normalize domain to lowercase to prevent case-variance bypass
    string memory normalizedDomain = _toLowercase(agentDomain);

    // Check for duplicates using normalized domain
    if (_domainToAgentId[normalizedDomain] != 0) {
        revert DomainAlreadyRegistered();
    }
    if (_addressToAgentId[agentAddress] != 0) {
        revert AddressAlreadyRegistered();
    }

    // Assign new agent ID
    agentId = _agentIdCounter++;

    // Store agent info with original domain (for display) but use normalized for lookups
    _agents[agentId] = AgentInfo({
        agentId: agentId,
        agentDomain: agentDomain, // Store original case for display
        agentAddress: agentAddress
    });

    // Create lookup mappings using normalized domain
    _domainToAgentId[normalizedDomain] = agentId;
    _addressToAgentId[agentAddress] = agentId;

    emit AgentRegistered(agentId, agentDomain, agentAddress);
}
```

While the function prevents impersonation by requiring that `msg.sender == agentAddress`, it doesn't protect against front-running attacks on domain names. An attacker could see a pending transaction for a valuable domain name and submit their own transaction with a higher gas price to register that domain first. This is similar to how MEV bots front-run DeFi transactions to extract value.

To mitigate this, the contract could implement a commit-reveal scheme for domain registration or use other front-running protection mechanisms.

5 of 11 Findings

src/IdentityRegistry.sol

ASCII-only case insensitivity in domain normalization

Low Risk

The `_toLowercase()` function only converts ASCII uppercase letters (A-Z) to lowercase, not handling Unicode characters with case distinctions:

```
function _toLowercase(string memory str) internal pure returns (string memory result) {
    bytes memory strBytes = bytes(str);
    bytes memory resultBytes = new bytes(strBytes.length);

    for (uint256 i = 0; i < strBytes.length; i++) {
        // Convert A-Z to a-z
        if (strBytes[i] >= 0x41 && strBytes[i] <= 0x5A) {
            resultBytes[i] = bytes1(uint8(strBytes[i]) + 32);
        } else {
            resultBytes[i] = strBytes[i];
        }
    }

    result = string(resultBytes);
}
```

If domains can include non-ASCII characters (which is increasingly common with internationalized domain names), this could potentially allow for domain collisions. For example, Unicode characters like 'É' (U+00C9) and 'é' (U+00E9) would not be normalized to the same representation, potentially allowing similar-looking domains to be registered separately.

6 of 11 Findings

src/ValidationRegistry.sol

Event Emission Griefing in `validationRequest`

Low Risk

The `validationRequest` function allows anyone to submit a validation request. If a request for a specific `dataHash` is already pending (not expired and not responded to), the function permits anyone to call it again with the same parameters (`agentValidatorId`, `agentServerId`, `dataHash`). This action does not change the on-chain state but re-emits the `ValidationRequestEvent`.

```
// Check if request already exists and is still valid
IValidationRegistry.Request storage existingRequest = _validationRequests[dataHash];
if (existingRequest.dataHash != bytes32(0)) {
    if (block.timestamp <= existingRequest.timestamp + EXPIRATION_TIME) {
        if (existingRequest.agentValidatorId == agentValidatorId &&
            existingRequest.agentServerId == agentServerId) {
            // Event is re-emitted on every matching call
            emit ValidationRequestEvent(agentValidatorId, agentServerId, dataHash);
        }
        return;
    }
    // ...
}
```

Because the protocol relies heavily on off-chain indexers listening to events, this behavior can be exploited as a griefing vector. An attacker could repeatedly call the function for a legitimate pending request, flooding off-chain services with redundant events. This could cause unnecessary processing load, increase operational costs for indexers, and potentially disrupt services that depend on this event stream.

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

PUSH0 Opcode Compatibility Issue

Low Risk

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

```
pragma solidity ^0.8.19;
```

This can cause deployment failures on blockchain networks that haven't implemented the Shanghai upgrade or don't support the PUSH0 opcode, such as certain Layer 2 solutions or sidechains.

To ensure compatibility across different networks, either:

1. Lock the pragma to a specific version before 0.8.20 (e.g., `pragma solidity 0.8.19;`)
2. Explicitly set the EVM version in your compiler settings if using 0.8.20 or higher
3. Ensure that your target deployment networks support the Shanghai EVM version

8 of 11 Findings src/IdentityRegistry.sol

Inefficient String Operations

Info

The `_toLowercase` function in the IdentityRegistry contract creates a new string for domain normalization, which could be gas-intensive for long domains. This operation happens in both `newAgent` and `resolveByDomain` functions, potentially increasing gas costs for these operations. A more gas-efficient approach could be to perform case-insensitive comparisons without creating new strings, or to limit the maximum domain length to prevent excessive gas consumption.

9 of 11 Findings

src/ValidationRegistry.sol

Potential Front-Running in Validation Requests

Info

The `validationRequest` function in the ValidationRegistry contract is susceptible to front-running attacks where an attacker could monitor the mempool for validation request transactions and submit their own transaction with a higher gas price to request validation for the same data hash first.

```
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();
    }

    // SECURITY: Prevent self-validation to maintain validation integrity
    if (agentValidatorId == agentServerId) {
        revert SelfValidationNotAllowed();
    }

    // Check if request already exists and is still valid
    IValidationRegistry.Request storage existingRequest = _validationRequests[dataHash];
    if (existingRequest.dataHash != bytes32(0)) {
        if (block.timestamp <= existingRequest.timestamp + EXPIRATION_TIME) {
            // SECURITY: Only emit event if the request is from the same validator
            // Prevent misleading events from different validators
            if (existingRequest.agentValidatorId == agentValidatorId &&
                existingRequest.agentServerId == agentServerId) {
                emit ValidationRequestEvent(agentValidatorId, agentServerId, dataHash);
            }
            return;
        } else {
            // SECURITY: Clear stale response data when request expires
            delete _validationResponses[dataHash];
        }
    }

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

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

The function checks if a validation request already exists for a given data hash, but it doesn't verify that the request came from a specific agent or address. This means that an attacker could front-run a legitimate validation request by submitting their own request for the same data hash with different agent IDs. This could disrupt the validation process and potentially lead to incorrect validations.

To mitigate this, the contract could implement additional checks to ensure that validation requests can only be made by authorized parties, or use a commit-reveal scheme to prevent front-running.

10 of 11 Findings

src/ReputationRegistry.sol

No Mechanism to Revoke Feedback Authorizations

Info

Once a feedback authorization is created in the ReputationRegistry, it remains valid indefinitely. There is no mechanism for a server agent to revoke an authorization if the relationship with the client changes or if the authorization was created in error.

```
function acceptFeedback(uint256 agentClientId, uint256 agentServerId) external {
    // ... validation checks ...

    // Generate unique feedback authorization ID
    bytes32 feedbackAuthId = _generateFeedbackAuthId(agentClientId, agentServerId);

    // Store the authorization
    _feedbackAuthorizations[feedbackAuthId] = true;
    _clientServerToAuthId[agentClientId][agentServerId] = feedbackAuthId;

    emit AuthFeedback(agentClientId, agentServerId, feedbackAuthId);
}
```

Without a revocation mechanism, server agents have no way to control feedback authorizations once they've been granted. This could be problematic in scenarios where a client-server relationship deteriorates, or if a client should no longer be allowed to provide feedback. The permanent nature of these authorizations could lead to unwanted feedback from previously authorized clients.

11 of 11 Findings

src/ReputationRegistry.sol src/ValidationRegistry.sol

No Batch Operations

Best Practices

The contracts do not provide mechanisms for batch operations, such as authorizing feedback for multiple clients or making validation requests for multiple data hashes in a single transaction. This could lead to higher gas costs and more complex client-side implementations when dealing with multiple operations. Adding batch functions could improve gas efficiency and user experience for applications that need to perform multiple operations.

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.