Scanned Code Report

AUDITAGENT

Powered by NETHERMIND SECURITY

Code Info

Deep Scan

Scan ID 2	Date August 30, 2025
Organization ChaosChain	Repository trustless-agents-erc-ri
Branch main	Commit Hash 052cc8d2...7c1f2840

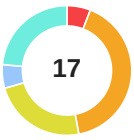
Contracts in scope

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

Code Statistics

Findings 17	Contracts Scanned 3	Lines of Code 480

Findings Summary



Total Findings

- High Risk (1)
- Medium Risk (7)
- Low Risk (4)
- Info (1)
- Best Practices (4)

Code Summary

This protocol provides a comprehensive framework for managing decentralized identities, reputation, and data validation for autonomous agents, structured around three core registries.

1. **IdentityRegistry**: This contract serves as the central database for agent identities. It allows for the creation of new agent profiles by linking a unique domain name and an Ethereum address to a unique numerical agent ID. This registry ensures that each agent has a distinct and verifiable on-chain identity. Agents have the autonomy to update their own registered information, such as their domain or associated address.

2. **ReputationRegistry**: Building upon the identity layer, this contract facilitates a lightweight, on-chain mechanism for authorizing feedback between agents. A "server" agent that has provided a service can explicitly authorize a "client" agent to submit feedback. This process generates a unique authorization record, which can then be leveraged by higher-level reputation systems (either on-chain or off-chain) to build trust scores.

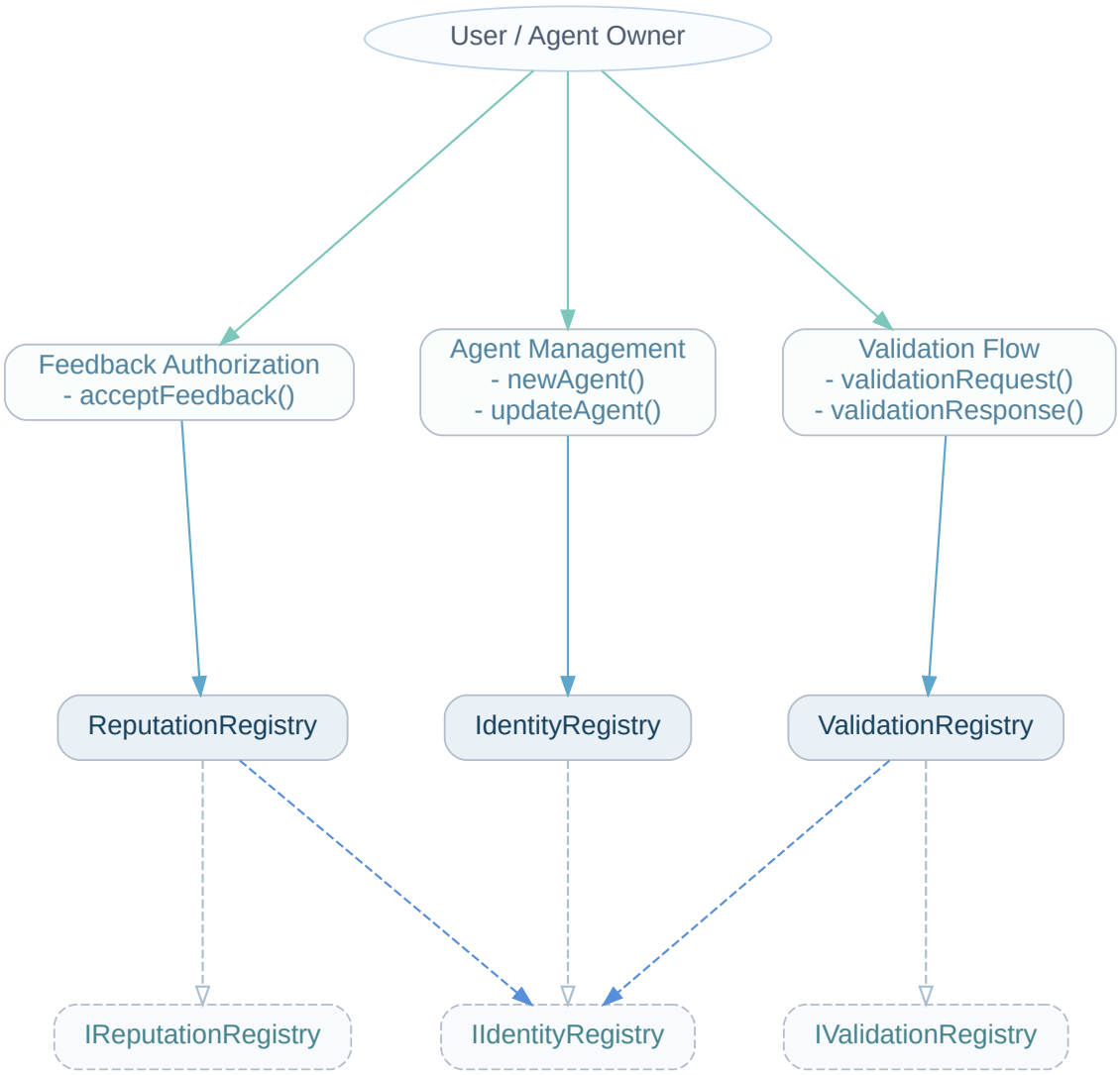
3. **ValidationRegistry**: This contract establishes a formal process for independent data validation. An agent can request that a specific "validator" agent review a piece of data, identified by its hash. The designated validator can then submit a quantitative response (a score from 0-100) within a defined time window. This creates a verifiable, third-party attestation about the validity or quality of the data.

Together, these three components form a foundational infrastructure for trustless agent-based systems, enabling verifiable identity, authorized feedback loops, and independent data validation.

Entry Points and Actors

- `newAgent(string calldata agentDomain, address agentAddress)`: Anyone can register a new agent by providing a unique domain and address, which generates a new agent ID.
- `updateAgent(uint256 agentId, string calldata newAgentDomain, address newAgentAddress)`: The owner of an agent (the address associated with the agent ID) can update its registered domain or address.
- `acceptFeedback(uint256 agentClientId, uint256 agentServerId)`: A server agent's owner can authorize a client agent to provide feedback, creating a verifiable authorization record.
- `validationRequest(uint256 agentValidatorId, uint256 agentServerId, bytes32 dataHash)`: Anyone can initiate a request for a specific validator agent to review data associated with a server agent.
- `validationResponse(bytes32 dataHash, uint8 response)`: The designated validator agent's owner can submit a response to a pending validation request.

Code Diagram



1 of 17 Findings — src/IdentityRegistry.sol

Missing ownership check enables arbitrary address registration — High Risk

The newAgent function accepts an agentAddress parameter but does not require msg.sender to match or prove control over that address. Any user can therefore create a registry entry for any external address, associating it with an arbitrary domain. This prevents the legitimate address owner from registering themselves (AddressAlreadyRegistered revert) and may mislead integrators that rely on the registry, resulting in impersonation or denial of service.

2 of 17 Findings — src/IdentityRegistry.sol

Registration fee advertised in the documentation is never enforced – unlimited, cost-free agent creation enables state-bloat DoS — Medium Risk

The white-paper specifies a 0.005 ETH fee that is "burned" for every `newAgent` call to discourage spam registrations.
In the actual implementation `newAgent` is declared *non-payable* and contains no `require(msg.value == …)` check or `burn` logic.

```
function newAgent(string calldata agentDomain, address agentAddress)
        external // <- **non-payable**
        returns (uint256 agentId)
{
    …                           // no msg.value validation
}
```

Because creating an agent is completely free, any user (or bot network) can register an unbounded number of domains/addresses and permanently fill the `_agents`, `_domainToAgentId`, and `_addressToAgentId` mappings. Each entry consumes between ~2,600 and ~20,000 gas of persistent storage; millions of registrations would make every subsequent write more expensive and could eventually price legitimate users out of the system. The registry has no pruning mechanism, so the attack is irreversible once executed.

3 of 17 Findings

src/IdentityRegistry.sol

Front-Running Vulnerability in Agent Registration

Medium Risk

The `newAgent` function is susceptible to front-running attacks. An attacker monitoring the mempool could see a transaction to register a specific domain/address and front-run it by submitting their own transaction with the same domain/address but a higher gas price.

```
function newAgent(
    string calldata agentDomain,
    address agentAddress
) external returns (uint256 agentId) {
    // Validate inputs
    if (bytes(agentDomain).length == 0) {
        revert InvalidDomain();
    }
    if (agentAddress == address(0)) {
        revert InvalidAddress();
    }

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

    // Assign new agent ID
    agentId = _agentIdCounter++;
    //... registration logic continues
}
```

This could allow attackers to "steal" desirable domain names from legitimate users or prevent specific addresses from registering. Since the registry is meant to be a core identity system for agents, this could undermine trust in the protocol by allowing malicious actors to impersonate others or register domains that should belong to legitimate users.

4 of 17 Findings

src/IdentityRegistry.sol

Domain uniqueness bypass via case-variance allows impersonation

Medium Risk

`IdentityRegistry` enforces uniqueness by storing the raw string that is supplied by a registrant:

```
mapping(string => uint256) private _domainToAgentId;
...
if (_domainToAgentId[agentDomain] != 0) revert DomainAlreadyRegistered();
```

Because string comparison in Solidity is byte-wise, the registry treats `"EXAMPLE.com"`, `"example.com"` and `"Example.com"` as **different keys**. A malicious user can therefore register a visually similar (or Unicode-confusable) domain and mislead clients that rely on `resolveByDomain`. Nothing in the contract normalises the input (lower-casing, puny-code conversion, UTF-8 NFC, …), so the *global uniqueness invariant claimed by the specification is silently broken*.

Practical impact: a malicious agent can front-run the legitimate registration of `example.com` by using `ExAmPlE.com`, appear in discovery services, and later abuse the confusion to collect feedback/validation meant for the real agent.

5 of 17 Findings

src/ValidationRegistry.sol

State Pollution from Stale Validation Responses

Medium Risk

The `ValidationRegistry` contract does not properly clear stale response data when a new validation request overwrites an expired one for the same `dataHash`. The contract uses separate mappings (`_validationResponses`, `_hasResponse`) to store response data, which are not reset when a new request is created for a previously used `dataHash`.

This leads to a critical state inconsistency. If a request for a `dataHash` is made, responded to, and then expires, a new request for the same `dataHash` can be created. However, calling `getValidationResponse()` for this `dataHash` will return the old, stale response from the previous request, even though the new request is still pending. This breaks the integrity of the validation system by incorrectly associating old data with a new, unrelated request.

Vulnerable Code Snippet:

```
// File: src/ValidationRegistry.sol

function validationRequest(
    uint256 agentValidatorId,
    uint256 agentServerId,
    bytes32 dataHash
) external {
    // ...
    IValidationRegistry.Request storage existingRequest = _validationRequests[dataHash];
    if (existingRequest.dataHash != bytes32(0)) {
        if (block.timestamp <= existingRequest.timestamp + EXPIRATION_TIME) {
            // ...
            return;
        }
        // The old request is expired, but its response data in _validationResponses
        // and _hasResponse is NOT cleared before overwriting the request.
    }

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

    emit ValidationRequestEvent(agentValidatorId, agentServerId, dataHash);
}

function getValidationResponse(bytes32 dataHash) external view returns (bool hasResponse,
uint8 response) {
    // This function reads from _hasResponse and _validationResponses, which may contain
stale data.
    hasResponse = _hasResponse[dataHash];
    if (hasResponse) {
        response = _validationResponses[dataHash];
    }
}
```

This flaw allows stale data to be presented as valid, undermining the reliability of the validation mechanism.

6 of 17 Findings

src/ValidationRegistry.sol

Validation requests can be hijacked by front-running, blocking the legitimate validator • Medium Risk

Anyone can submit a `validationRequest()` for any `dataHash`. The first request stored for a given hash is preserved until it expires, and subsequent calls merely re-emit an event without changing the stored validator:

```
if (existingRequest.dataHash != bytes32(0)) {
    if (block.timestamp <= existingRequest.timestamp + EXPIRATION_TIME) {
        emit ValidationRequestEvent(agentValidatorId, agentServerId, dataHash);
        return;              // state unchanged
    }
}
```

An attacker who observes a pending transaction can front-run it with the **same `dataHash` but a validator ID they control**. The genuine transaction then finds an existing request and does not overwrite it. Consequently, when the real validator later calls `validationResponse`, the call reverts with `UnauthorizedValidator`, and the server's validation is effectively blocked for the whole 1 000-second window. The attacker can keep repeating this just before each expiry, creating a perpetual denial-of-service against any data hash.

7 of 17 Findings

src/ValidationRegistry.sol

Misleading Event Emission on Existing Validation Request

Medium Risk

The `validationRequest` function in `ValidationRegistry.sol` contains a flaw when handling a call for a `dataHash` that already has a pending (non-expired) request. In this case, the function emits a `ValidationRequestEvent` with the new `agentValidatorId` and `agentServerId` provided in the function call, but it does not update the on-chain state of the existing request. This creates a critical inconsistency where the event log suggests a request has been created or updated with new participants, while the contract's storage still holds the original data. Given that the protocol is designed to be event-driven for off-chain systems, this misleading event can cause off-chain consumers to have an incorrect state, leading to failed interactions and breaking the trust layer of the protocol.

```
// File: src/ValidationRegistry.sol

function validationRequest(
    uint256 agentValidatorId,
    uint256 agentServerId,
    bytes32 dataHash
) external {
    // ...
    IValidationRegistry.Request storage existingRequest = _validationRequests[dataHash];
    if (existingRequest.dataHash != bytes32(0)) {
        if (block.timestamp <= existingRequest.timestamp + EXPIRATION_TIME) {
            // Flaw: Emits event with new parameters from the call (agentValidatorId,
agentServerId)
            // but the stored request (existingRequest) is not modified.
            emit ValidationRequestEvent(agentValidatorId, agentServerId, dataHash);
            return;
        }
    }
    // ...
}
```

8 of 17 Findings — src/ValidationRegistry.sol

Risky Strict Equality Check

Medium Risk

The contract uses a dangerous strict equality check when comparing a hash value to zero.

In `ValidationRegistry.getValidationRequest(bytes32)` at line 138, the code checks:

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

This strict equality check is problematic because it's used to determine if a validation request exists. If the request doesn't exist, the function will revert. However, using strict equality with zero values can lead to false negatives in certain edge cases.

Consider using a more robust existence check or implementing a proper mapping with boolean flags to track existence of validation requests.

9 of 17 Findings — src/ValidationRegistry.sol

Unbounded state growth in ValidationRegistry

Low Risk

Neither expired nor responded validation requests are ever removed from storage:

```
// No deletion in validationResponse()
request.responded = true;
_validationResponses[dataHash] = response;
```

and `validationRequest()` only overwrites a slot when the **same** `dataHash` is used after expiration. A malicious actor can continuously submit unique `dataHash` values, each creating new `Request` and response slots that stay in storage permanently. Over time this leads to unbounded state growth, increasing contract maintenance costs and potentially making some functions more expensive due to larger storage reads.

10 of 17 Findings

src/ValidationRegistry.sol

Validation request can be created with same agent as validator and server

Low Risk

The `validationRequest` function in the ValidationRegistry contract does not prevent the same agent from being specified as both the validator and the server. This allows a validation request to be created where an agent validates their own data, which defeats the purpose of independent validation.

```
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();
    }

    // Missing check: if (agentValidatorId == agentServerId) revert SameAgentValidation();

    // ... rest of the function
}
```

The function should include a check to ensure that `agentValidatorId` is not equal to `agentServerId` to maintain the integrity of the validation system. Without this check, an agent could create validation requests for themselves and potentially respond to those requests, creating a false impression of independent validation.

11 of 17 Findings src/ReputationRegistry.sol

Permanent Feedback Authorizations Without Revocation

Low Risk

Once a feedback authorization is created in the `ReputationRegistry`, it never expires and cannot be revoked. This could be problematic if the relationship between agents changes over time, as there's no way to prevent a client from providing feedback indefinitely after being authorized once.

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

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

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

    emit AuthFeedback(agentClientId, agentServerId, feedbackAuthId);
    // No mechanism to revoke or expire this authorization
}
```

The lack of expiration or revocation mechanisms means that once a server agent authorizes a client for feedback, this authorization persists indefinitely. If the client later becomes malicious or if the server no longer wishes to receive feedback from that client, there is no on-chain way to prevent the client from continuing to provide feedback.

12 of 17 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.

This presents a compatibility risk when deploying to networks that haven't implemented the Shanghai upgrade, particularly Layer 2 solutions or alternative EVM-compatible blockchains. On these networks, deployment will fail due to the unrecognized opcode.

To mitigate this issue:

1. Lock the pragma to a specific version before 0.8.20 (e.g., `pragma solidity 0.8.19;`), or
2. If using 0.8.20+, explicitly set the EVM version in your compiler settings to a pre-Shanghai version.

13 of 17 Findings src/ValidationRegistry.sol

Inconsistent Expiration Mechanism: Seconds Implemented vs. Blocks Documented

Info

There is a significant discrepancy between the documented expiration mechanism for validation requests and the actual implementation in the `ValidationRegistry` contract. The documentation and EIP summary consistently refer to a "1000-block expiration" window. However, the contract's implementation uses `block.timestamp` and a constant `EXPIRATION_TIME` of 1000, which corresponds to 1000 seconds (approximately 16.7 minutes), not 1000 blocks (approximately 3.3 hours on Ethereum Mainnet).

This inconsistency can mislead developers, integrators, and users of the protocol, causing them to make incorrect assumptions about the time-sensitivity of validation requests. For example, a validator might believe they have more time to respond than is actually available, leading to missed validations.

Relevant Code:

```
// File: src/ValidationRegistry.sol

/// @dev Expiration time for validation requests (in seconds)
uint256 public constant EXPIRATION_TIME = 1000;

// ...

function validationResponse(bytes32 dataHash, uint8 response) external {
    // ...
    // Check if request has expired using block.timestamp
    if (block.timestamp > request.timestamp + EXPIRATION_TIME) {
        revert RequestExpired();
    }
    // ...
}
```

The function `getExpirationSlots()` is also misleadingly named, as it returns a duration in seconds, not blocks or "slots".

14 of 17 Findings src/IdentityRegistry.sol src/ReputationRegistry.sol src/ValidationRegistry.sol

Lack of explicit versioning mechanism

Best Practices

None of the contracts implement an explicit versioning mechanism to track implementation changes. As the protocol evolves, this could make it difficult to track which version of the implementation is deployed and to ensure compatibility between different components. Adding version tracking would aid in governance, upgrades, and integration with external systems.

15 of 17 Findings

src/ValidationRegistry.sol

Inconsistent and Redundant State for Validation Responses

Best Practices

The `ValidationRegistry` contract uses two separate state variables to track whether a validation request has been fulfilled: the `responded` boolean within the `Request` struct stored in the `_validationRequests` mapping, and a separate boolean mapping `_hasResponse`. Both variables are written to in the `validationResponse` function.

This design is redundant, as a single source of truth would suffice. It increases the gas cost for each `validationResponse` transaction due to an additional `SSTORE` operation. Furthermore, it leads to inconsistent read patterns within the contract: the `isValidationPending` function checks `request.responded`, while the `getValidationResponse` function checks `_hasResponse`. This redundancy complicates the contract's logic and increases maintenance overhead.

```
// in validationResponse(...)
        // Mark as responded and store the response
        request.responded = true; // First state update
        _validationResponses[dataHash] = response;
        _hasResponse[dataHash] = true; // Second, redundant state update
```

16 of 17 Findings

src/ReputationRegistry.sol

Use of Deprecated `block.difficulty` Opcode

Best Practices

The `_generateFeedbackAuthId` function in `ReputationRegistry.sol` uses `block.difficulty` as a source of entropy for generating a unique ID. Following the Ethereum Merge, the `DIFFICULTY` opcode (0x44) was repurposed and now returns the output of the `PREVRANDAO` opcode, as specified in EIP-4399. While it still provides a source of randomness, relying on the deprecated `block.difficulty` is not a best practice. For code clarity, forward-compatibility, and to align with modern Solidity standards, it is recommended to use `block.prevrandao` directly.

```
// File: src/ReputationRegistry.sol

function _generateFeedbackAuthId(
    uint256 agentClientId,
    uint256 agentServerId
) private view returns (bytes32 feedbackAuthId) {
    // Include block timestamp and transaction hash for uniqueness
    feedbackAuthId = keccak256(
        abi.encodePacked(
            agentClientId,
            agentServerId,
            block.timestamp,
            block.difficulty, // Use of deprecated opcode
            tx.origin
        )
    );
}
```

17 of 17 Findings

src/ReputationRegistry.sol

Use of `tx.origin` leaks unnecessary information and breaks meta-transaction compatibility

Best Practices

The hash that builds `feedbackAuthId` inserts `tx.origin`, which (1) needlessly exposes the EOAs behind smart-wallet calls and (2) causes the generated ID to change when relayed through a meta-transaction hub (where `tx.origin` ≠ end-user). Relying on `tx.origin` is discouraged by Solidity best-practices.

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.