Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: 9 November 2026

M. De Gennaro
Stalwart Labs

8 May 2026

MTA Hooks: An HTTP-Based Mail Processing Protocol
draft-degennaro-mta-hooks-01

Abstract

This document specifies MTA Hooks, an HTTP-based protocol enabling Mail Transfer Agents (MTAs) to delegate message processing decisions to external services. MTA Hooks provides a modern alternative to legacy mail filtering protocols by leveraging ubiquitous HTTP infrastructure, supporting both JSON and CBOR serialization, and offering fine-grained capability negotiation. The protocol supports both inbound message reception and outbound message delivery scenarios, allowing external scanners to inspect messages, modify content, and influence routing decisions at various stages of mail processing.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 9 November 2026.

Copyright Notice

Table of Contents

1. Introduction

Mail Transfer Agents require extensible mechanisms to integrate with external services for spam filtering, virus scanning, policy enforcement, and compliance monitoring. While legacy protocols such as the Milter protocol have served this purpose, they present challenges in modern deployments including proprietary wire formats, limited tooling, and operational complexity.

MTA Hooks addresses these challenges by defining an HTTP-based protocol for mail processing delegation. The protocol leverages the widespread availability of HTTP client and server implementations, standard serialization formats, and established operational practices for HTTP services.

MTA Hooks achieves broad deployment compatibility by building upon standard HTTP semantics and methods, allowing implementers to leverage existing HTTP client and server libraries. The protocol supports both JSON and CBOR serialization to accommodate environments with different performance and parsing requirements. Capability negotiation during registration enables graceful feature discovery and forward compatibility as the protocol evolves. Mandatory transport encryption protects message content in transit, while the authentication mechanism remains pluggable to integrate with diverse deployment environments. The HTTP foundation ensures compatibility with existing operational infrastructure including load balancers, reverse proxies, and monitoring systems.

MTA Hooks supports two primary use cases: inbound processing during message reception from remote clients, and outbound processing during message delivery to remote servers. Both scenarios follow a consistent request-response pattern where the MTA invokes the registered hook endpoints at configured processing stages.

1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.2. Terminology

The following terms are used throughout this document:

* MTA (Mail Transfer Agent): A software component responsible for transferring electronic mail messages between hosts, as described in [RFC5321].

* Hook Endpoint: An HTTP server endpoint that receives hook invocations from an MTA and returns processing instructions.

* Scanner: A service that registers with an MTA to receive hook invocations. Scanners perform functions such as spam filtering, virus scanning, or policy enforcement. The terms "scanner" and "hook endpoint" are used interchangeably when referring to the service receiving hook requests.

* Registration: The process by which an MTA establishes a session with a scanner, including capability negotiation and assignment of a registration identifier.

* Inbound Processing: Hook invocation during message reception, when the MTA accepts a message from a remote SMTP client.

* Outbound Processing: Hook invocation during message delivery, when the MTA transmits a message to a remote SMTP server.

* Stage: A specific point in mail processing at which the MTA invokes registered hooks. Different stages provide access to different message properties.

* Action: An instruction from a scanner indicating how the MTA should proceed with message processing.

* Modification: A change to message properties requested by a scanner, expressed as operations on the hook request object.

1.3. Data Types

This specification defines several object types used throughout the protocol. The following primitive types are used:

String
: A JSON string value.

Int
: An integer in the range -2^53+1 <= value <= 2^53-1.

UnsignedInt
: An integer in the range 0 <= value <= 2^53-1.

Boolean
: A JSON boolean value (true or false).

UTCDate
: A string in "date-time" format as defined in [RFC3339], where the time-offset component MUST be "Z" (UTC time). For example, "2024-12-21T15:30:00Z".

2. Protocol Overview

MTA Hooks defines a request-response protocol where the MTA acts as an HTTP client and scanners act as HTTP servers. During mail processing, the MTA constructs a request containing message properties and context information, transmits it to registered scanner endpoints, and processes the response to determine subsequent actions.

2.1. Architecture

The protocol architecture consists of three primary components:

1. The MTA, which initiates HTTP requests to scanner endpoints at configured processing stages.

2. Scanner services, which receive requests, perform analysis, and return responses containing actions and modifications.

3. A registration mechanism through which the MTA declares its intent to send specific stages and properties and the scanner returns a registration identifier used on subsequent invocations.

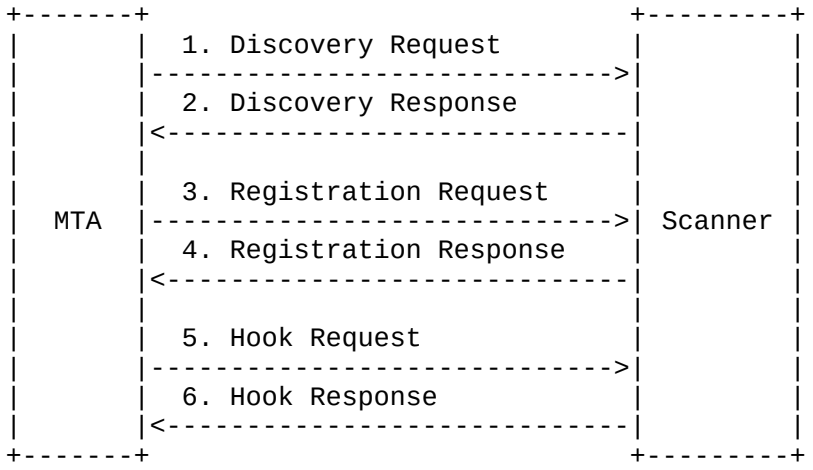

Figure 1: MTA Hooks Protocol Flow

The protocol flow proceeds as follows:

1. The MTA discovers scanner capabilities by requesting the well-known discovery endpoint.

2. The scanner returns a discovery document describing supported stages, actions, and configuration options.

3. The MTA submits a registration request to the scanner specifying the stages, properties, and filters it will deliver.

4. The scanner validates the request, returns a registration identifier, and confirms the negotiated capabilities.

5. During mail processing, the MTA invokes the scanner’s hook endpoint with message properties and context.

6. The scanner returns a response containing an action and optional modifications.

2.2. Serialization

Requests and responses are serialized using either JSON [RFC8259] or CBOR [RFC8949]. The serialization format is negotiated during registration and indicated via HTTP Content-Type headers.

For JSON serialization, the Content-Type header MUST be "application/json". For CBOR serialization, the Content-Type header MUST be "application/cbor".

Binary data handling differs between formats. In JSON serialization, binary content such as message body parts MUST be encoded using Base64. In CBOR serialization, binary content SHOULD be transmitted as raw byte strings.

2.3. Processing Stages

MTA Hooks defines processing stages for both inbound and outbound message handling. Scanners subscribe to specific stages during registration and receive invocations only for subscribed stages.

2.3.1. Inbound Stages

Inbound stages correspond to SMTP protocol events during message reception:

* connect: Invoked when a remote client establishes a connection, before any SMTP commands are exchanged.

* ehlo: Invoked after the client sends EHLO or HELO command.

* mail: Invoked after each MAIL FROM command.

* rcpt: Invoked after each RCPT TO command. This stage may be invoked multiple times per transaction.

* data: Invoked after the complete message has been received, following the DATA command and message content.

2.3.2. Outbound Stages

Outbound stages correspond to delivery events during message transmission:

* delivery: Invoked after a delivery attempt completes for all recipients, regardless of success or failure.

* defer: Invoked only when at least one recipient delivery cannot be completed and requires retry.

* dsn: Invoked when the MTA generates a Delivery Status Notification of any type.

For outbound stages, the hook is invoked once after delivery attempts for all recipients in a given delivery batch have completed. The scanner receives the delivery status for each recipient and may modify per-recipient handling.

2.4. Inbound Message Processing

During inbound processing, the MTA invokes registered hooks as it receives messages from remote SMTP clients. This corresponds to traditional mail filtering scenarios where external services inspect incoming mail for spam, viruses, or policy violations.

The inbound flow proceeds through SMTP protocol stages:

1. A remote client connects to the MTA.

2. The MTA invokes hooks registered for the "connect" stage.

3. The client sends EHLO/HELO; the MTA invokes "ehlo" stage hooks.

4. The client sends MAIL FROM; the MTA invokes "mail" stage hooks.

5. The client sends one or more RCPT TO commands; the MTA invokes "rcpt" stage hooks for each.

6. The client sends DATA and message content; the MTA invokes "data" stage hooks.

At each stage, scanners may instruct the MTA to accept, reject, or modify the transaction.

2.4.1. Multiple Scanner Handling

When multiple scanners are registered for the same stage, the MTA invokes them sequentially in an implementation-defined order. Each scanner’s response affects the request seen by subsequent scanners.

The scanner chain terminates early when a scanner returns a terminal action:

* Inbound terminal actions: "reject", "discard", "disconnect"

* Outbound terminal actions: "cancel"

Non-terminal actions ("accept", "quarantine" for inbound; "continue" for outbound) allow the chain to proceed. Each scanner sees the prior scanner’s modifications, including any change to the "/action" path, and MAY further modify it.

Action precedence within a chain proceeds from least to most restrictive: "accept" is the weakest, "quarantine" is stronger, and the terminal actions ("reject", "discard", "disconnect") are the strongest. By default, the MTA SHOULD reject responses that

downgrade the action to a weaker value (for example, setting "/action" from "quarantine" to "accept"). Implementations MAY provide configuration to permit such downgrades for specific trusted scanners, but this MUST NOT be the default.

Implementations SHOULD provide configuration options for:

* Scanner invocation order (priority-based or explicit ordering)

* Per-scanner trust to permit action downgrades

* Logging of chain termination events for operational visibility

Example chain behavior:

1. Scanner A (spam filter) sets action to "accept" and adds X-Spam-Score header

2. Scanner B (virus scanner) receives modified request, detects malware, sets action to "reject"

3. Chain terminates; Scanner C is not invoked

4. MTA rejects the message

2.5. Outbound Message Processing

During outbound processing, the MTA invokes registered hooks as it delivers messages to remote SMTP servers. This supports use cases including delivery logging, compliance monitoring, and routing decisions.

The outbound flow proceeds as follows:

1. The MTA selects a queued message for delivery and identifies the recipients due for delivery at this time.

2. The MTA attempts delivery to all selected recipients.

3. After all attempts in this delivery job complete, the MTA invokes hooks registered for the "delivery" stage.

4. If any recipient requires retry, the MTA invokes "defer" stage hooks.

5. If the MTA generates a DSN, it invokes "dsn" stage hooks before sending.

A delivery job represents a single processing cycle where the MTA retrieves a message from the queue and attempts delivery to one or more recipients. The specific batching of recipients into delivery jobs is implementation-defined and may depend on factors such as destination domain, connection reuse, or queue configuration. The "delivery" stage hook is invoked once per delivery job after all recipient attempts within that job have completed.

Scanners may modify per-recipient delivery status, suppress DSN generation, or alter retry scheduling.

2.6. Modifications

Scanners communicate changes through modifications to the hook request object, expressed as set, add, and delete operations on JSON Pointers [RFC6901]. The complete modification semantics, including operation ordering and error handling, are defined in Section 6.

3. Discovery

MTAs discover scanner capabilities through a well-known HTTP endpoint. Discovery is OPTIONAL; scanner endpoints MAY alternatively be configured manually.

3.1. Well-Known Endpoint

Scanners that support discovery MUST expose a discovery document at the path "/.well-known/mta-hooks". The MTA retrieves this document using an HTTP GET request.

The discovery endpoint MUST support JSON serialization. Support for CBOR serialization is OPTIONAL.

3.2. Discovery Document Format

The discovery document is a JSON or CBOR object containing the following fields:

version: String
: The protocol version. This specification defines version "1.0".

endpoints: DiscoveryEndpoints
: URLs for protocol operations.

serialization: String[]
: Supported serialization formats. Valid values are "json" and "cbor".

capabilities: DiscoveryCapabilities
Supported protocol capabilities.

limits: DiscoveryLimits|null
Operational limits, or null if no specific limits are advertised.

extensions: String[]|null
Supported protocol extensions. Extension identifiers SHOULD use a reverse domain name prefix for vendor-specific extensions.

A *DiscoveryEndpoints* object has the following properties:

registration: String
URL path for submitting registration requests.

deregistration: String
URL path template for deregistration, with "{registration_id}" placeholder.

A *DiscoveryCapabilities* object has the following properties:

inbound: DirectionCapabilities|null Inbound processing capabilities, or null if inbound processing is not supported.

outbound: DirectionCapabilities|null Outbound processing capabilities, or null if outbound processing is not supported.

A *DirectionCapabilities* object has the following properties:

stages: String[]
Supported stages for this processing direction.

actions: String[]
Supported actions that scanners may request.

fetchProperties: String[]
JSON Pointers for properties the MTA can include in hook requests.

updateProperties: String[]
JSON Pointers for properties scanners may modify.

A *DiscoveryLimits* object has the following properties:

maxMessageSize: UnsignedInt|null Maximum message size in bytes the scanner accepts.

maxRegistrations: UnsignedInt|null Maximum concurrent registrations.

timeoutMs: UnsignedInt|null Default timeout in milliseconds for hook invocations.

The fetchProperties and updateProperties arrays contain JSON Pointers as defined in [RFC6901]. Each pointer identifies a property within the hook request structure that the MTA supports including or that scanners may modify. For example, "/envelope/from/address" refers to the sender’s email address within the envelope object.

3.2.1. Example Discovery Document

```
{
  "version": "1.0",

  "endpoints": {
    "registration": "/v1/hooks/register",
    "deregistration": "/v1/hooks/register/{registration_id}"
  },

  "serialization": ["json", "cbor"],

  "capabilities": {
    "inbound": {
      "stages": ["connect", "ehlo", "mail", "rcpt", "data"],
      "actions": ["accept", "reject", "discard",
                  "quarantine", "disconnect"],
      "fetchProperties": ["/envelope", "/message", "/rawMessage",
                          "/server", "/tls", "/auth", "/senderAuth",
                          "/client", "/stage", "/action",
                          "/timestamp","/response", "/protocol",
                          "/queue"],
      "updateProperties": ["/envelope", "/message", "/rawMessage",
                           "/action", "/response"]
    },
    "outbound": {
      "stages": ["delivery", "defer", "dsn"],
      "actions": ["continue", "cancel"],
      "fetchProperties": ["/envelope", "/message", "/rawMessage",
                          "/server", "/queue", "/stage", "/action",
                          "/timestamp", "/protocol"],
      "updateProperties": ["/envelope/to", "/action", "/message",
                           "/rawMessage"]
    }
  },

  "limits": {
    "maxMessageSize": 52428800,
    "maxRegistrations": 64,
    "timeoutMs": 30000
  },

  "extensions": ["x-vendor-feature"]
}
```

3.3. Manual Configuration

Discovery is OPTIONAL. Administrators MAY configure scanner endpoints manually, including capability restrictions and authentication credentials. Manual configuration may be necessary in environments where the well-known endpoint is not accessible or where policy requires explicit configuration.

4. Registration

The MTA registers with each configured scanner to negotiate capabilities and establish a session identifier used on subsequent hook invocations. Registration creates state at the scanner that remains active until explicit deregistration or expiration.

The MTA acts as the HTTP client throughout: it discovers scanner capabilities, submits the registration request, manages the lifecycle of the registration, and invokes hooks. The MTA does not expose any HTTP server for scanner-to-MTA communication. Scanner endpoints, credentials, and per-scanner subscription policy are provisioned at the MTA out of band; this specification does not define the provisioning channel.

4.1. Registration Request

The MTA submits a registration request via HTTP POST to the registration endpoint advertised in the scanner’s discovery document, or configured manually. The request body contains a JSON or CBOR object with the following fields:

name: String
: Human-readable name identifying the MTA instance, suitable for display in scanner administrative interfaces.

version: String|null
: MTA software version string.

timeoutMs: UnsignedInt|null
: Hook invocation timeout the MTA will enforce, in milliseconds. The scanner MAY use this value to size internal queues and processing budgets, but it does not affect MTA behavior.

expiresAt: UTCDate|null
: Requested registration expiration timestamp. The scanner MAY assign a different expiration based on policy.

serialization: String

Requested serialization format for hook requests. MUST be a value from the scanner's advertised serialization list.

inbound: StageSubscription|null
Inbound hook configuration. Omit or set to null if not subscribing to inbound stages.

outbound: StageSubscription|null
Outbound hook configuration. Omit or set to null if not subscribing to outbound stages.

filter: FilterOperator|FilterCondition|null
Filter the MTA will apply locally before invoking the scanner. Disclosed for transparency so the scanner can interpret the message stream it receives. Uses filter syntax from Section 5.5 of [RFC8620].

metadata: String[String]|null
Arbitrary key-value pairs for operational metadata.

A *StageSubscription* object has the following properties:

stages: String[]
Stages to subscribe to. MUST be a subset of the MTA's supported stages for this direction.

properties: String[]|null
JSON Pointers specifying message properties to receive. A value of null requests all supported properties.

At least one of "inbound" or "outbound" MUST be present and non-null in the registration request.

4.1.1. Filter Configuration

The filter field uses the FilterOperator and FilterCondition syntax defined in Section 5.5 of [RFC8620]. A FilterOperator combines multiple conditions using AND, OR, or NOT logic. A FilterCondition specifies criteria that a message must match.

Filter conditions follow Section 4.4.1 of [RFC8621] with the following exclusions. The conditions listed below depend on mailbox state, thread state, or message-arrival timing not applicable to MTA processing, and are NOT supported:

* inMailbox

* inMailboxOtherThan

* before
* after
* allInThreadHaveKeyword
* someInThreadHaveKeyword
* noneInThreadHaveKeyword
* hasKeyword
* notKeyword

All other filter conditions defined in Section 4.4.1 of [RFC8621] are permitted by this specification, including text-matching conditions such as "from", "to", "cc", "bcc", "subject", "body", "header", and "text", and the size conditions "minSize" and "maxSize". However, support for specific conditions is determined by the scanner implementation. If the registration request specifies a filter condition that the scanner does not support, the registration MUST be rejected with an UNSUPPORTED_FILTER error.

The following additional conditions are defined for envelope filtering:

envelopeFrom: String
: Matches if the MAIL FROM address matches the given value. The value MAY include wildcards using the "*" character, which matches zero or more characters.

envelopeTo: String
: Matches if any RCPT TO address matches the given value. The value MAY include wildcards using the "*" character, which matches zero or more characters.

Filters apply only to the "data" stage for content-based conditions (those examining message headers or body). For the "mail" and "rcpt" stages, only envelopeFrom and envelopeTo conditions are evaluated; other conditions are ignored for these stages. Messages or envelope commands not matching the filter do not trigger hook invocations.

4.1.2. Example Registration Request

```
POST /v1/hooks/register HTTP/1.1
Host: scanner.example.com
Content-Type: application/json
Authorization: Bearer eyJhbGciOiJIUzI1NiIs...

{
  "name": "Acme MTA mx1",
  "version": "2.1.0",

  "timeoutMs": 25000,

  "expiresAt": "2025-12-21T15:30:00Z",

  "serialization": "json",

  "inbound": {
    "stages": ["data"],
    "properties": ["/message", "/envelope", "/senderAuth"]
  },

  "outbound": {
    "stages": ["delivery"],
    "properties": null
  },

  "filter": {
    "operator": "OR",
    "conditions": [
      {"from": "*@external.example.com"},
      {"envelopeFrom": "*@partner.example.org"}
    ]
  },

  "metadata": {
    "operator": "acme-mail",
    "environment": "production",
    "instance": "mx1"
  }
}
```

4.2. Registration Response

Upon successful registration, the scanner returns HTTP status 201 Created with a response body containing:

registrationId: String

Unique identifier for this registration. The identifier MUST consist of ASCII alphanumeric characters plus hyphen (-), underscore (_), and colon (:), with a maximum length of 255 characters.

status: String
Registration status. Values are "active" or "suspended".

createdAt: UTCDate
Timestamp of registration creation.

expiresAt: UTCDate|null
Timestamp when registration expires, or null if no expiration is set.

hookEndpoint: String
Absolute or relative URL where the MTA MUST send hook invocations for this registration. A relative URL is resolved against the registration endpoint URL. The scanner MAY return a different path per registration to multiplex tenants.

negotiated: NegotiatedCapabilities
Final negotiated capabilities representing the intersection of the MTA’s request and the scanner’s support.

endpoints: RegistrationEndpoints
URLs for managing this registration.

A *NegotiatedCapabilities* object has the following properties:

serialization: String
Confirmed serialization format.

inbound: NegotiatedSubscription|null
Confirmed inbound configuration, or null if not registered for inbound processing.

outbound: NegotiatedSubscription|null
Confirmed outbound configuration, or null if not registered for outbound processing.

A *NegotiatedSubscription* object has the following properties:

stages: String[]
Confirmed stages.

properties: String[]
Confirmed property list.

A *RegistrationEndpoints* object has the following properties:

deregistration: String
URL for deregistration requests.

status: String
URL for status queries.

4.2.1. Example Registration Response

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: /v1/hooks/register/reg_7a3b9c2e-4d5f-6a7b-8c9d-0e1f2a3b4c5d

{
  "registrationId": "reg_7a3b9c2e-4d5f-6a7b-8c9d-0e1f2a3b4c5d",
  "status": "active",
  "createdAt": "2024-12-21T15:30:00Z",
  "expiresAt": "2025-12-21T15:30:00Z",

  "hookEndpoint": "/v1/hooks/invoke/reg_7a3b9c2e-4d5f-6a7b-8c9d-0e1f2a3b4c5d",

  "negotiated": {
    "serialization": "json",
    "inbound": {
      "stages": ["data"],
      "properties": ["/message", "/envelope", "/senderAuth"]
    },
    "outbound": {
      "stages": ["delivery"],
      "properties": ["/message", "/envelope", "/queue", "/server"]
    }
  },

  "endpoints": {
    "deregistration":
      "/v1/hooks/register/reg_7a3b9c2e-4d5f-6a7b-8c9d-0e1f2a3b4c5d",
    "status":
      "/v1/hooks/register/reg_7a3b9c2e-4d5f-6a7b-8c9d-0e1f2a3b4c5d/status"
  }
}
```

4.3. Authentication

The MTA MUST authenticate to the scanner on every registration, status, and deregistration request. The same credentials are used for hook invocations (Section 5.2). The specific mechanism is out of scope for this specification; implementations SHOULD support at least one of:

* Bearer tokens via the HTTP Authorization header
* Mutual TLS with client certificate verification
* HMAC-based pre-shared key authentication

Credentials are issued by the scanner operator to each MTA permitted to register, through an out-of-band provisioning channel. The scanner MUST verify the MTA's identity from the presented credentials and reject unauthenticated or unauthorized requests. See Section 9 for deployment guidance.

4.4. Registration Lifecycle

4.4.1. Status Queries

The MTA MAY query a registration's status via HTTP GET to the status endpoint returned during registration.

```
GET /v1/hooks/register/reg_7a3b9c2e-4d5f-6a7b-8c9d-0e1f2a3b4c5d/status \
  HTTP/1.1
Host: scanner.example.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIs...
```

The response contains registration status and optional statistics:

registrationId: String
: The registration identifier.

status: String
: Current status: "active", "suspended", or "deregistered".

createdAt: UTCDate|null
: Timestamp of creation.

expiresAt: UTCDate|null
: Timestamp of expiration.

lastInvocation: UTCDate|null
: Timestamp of most recent hook invocation.

statistics: RegistrationStatistics|null
Operational statistics.

health: HealthStatus|null
Health check status.

A *RegistrationStatistics* object has the following properties:

invocations: InvocationCounts
Invocation counts.

errors: InvocationCounts
Error counts with the same time period structure as invocations.

averageResponseMs: Number
Average response time in milliseconds.

An *InvocationCounts* object has the following properties:

total: UnsignedInt
Total count since registration.

last24h: UnsignedInt
Count in past 24 hours.

last7d: UnsignedInt
Count in past 7 days.

last30d: UnsignedInt
Count in past 30 days.

A *HealthStatus* object has the following properties:

status: String
Health status: "healthy", "degraded", or "unhealthy".

lastCheck: UTCDate
Timestamp of last health check.

consecutiveFailures: UnsignedInt
Count of consecutive failed health checks.

4.4.2. Example Status Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "registrationId": "reg_7a3b9c2e-4d5f-6a7b-8c9d-0e1f2a3b4c5d",
  "status": "active",
  "createdAt": "2024-12-21T15:30:00Z",
  "expiresAt": "2025-12-21T15:30:00Z",
  "lastInvocation": "2024-12-21T16:42:18Z",

  "statistics": {
    "invocations": {
      "total": 15482,
      "last24h": 3241,
      "last7d": 10234,
      "last30d": 43210
    },
    "errors": {
      "total": 32,
      "last24h": 5,
      "last7d": 12,
      "last30d": 28
    },
    "averageResponseMs": 45
  },

  "health": {
    "status": "healthy",
    "lastCheck": "2024-12-21T16:44:00Z",
    "consecutiveFailures": 0
  }
}
```

4.4.3. Registration Expiration and Renewal

Registrations MAY have an expiration time set by the scanner based on policy. The MTA MUST re-register before expiration to maintain continuous service. Re-registration follows the same process as initial registration and requires full capability renegotiation; the scanner SHOULD issue a fresh registration identifier on renewal and SHOULD continue to honor hook invocations carrying the prior identifier for a brief grace period to avoid races.

The MTA SHOULD monitor its registration status (either by tracking expiresAt locally or by polling the status endpoint) and initiate re-registration before expiration.

4.5. Deregistration

The MTA deregisters by sending an HTTP DELETE request to the deregistration endpoint.

```
DELETE /v1/hooks/register/reg_7a3b9c2e-4d5f-6a7b-8c9d-0e1f2a3b4c5d \
  HTTP/1.1
Host: scanner.example.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIs...
```

Upon successful deregistration, the scanner returns HTTP status 200 with confirmation:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "registrationId": "reg_7a3b9c2e-4d5f-6a7b-8c9d-0e1f2a3b4c5d",
  "status": "deregistered",
  "deregisteredAt": "2024-12-21T16:45:00Z"
}
```

After deregistration, the MTA MUST NOT send further hook invocations referencing the deregistered registration identifier.

4.5.1. In-Flight Requests

Hook invocations that are in-flight at the time of deregistration MAY complete normally. The scanner SHOULD accept and respond to hook requests that arrive during or shortly after deregistration processing, as network latency may cause requests dispatched before deregistration to arrive afterward.

Once deregistration completes, the MTA MUST NOT initiate new hook invocations to the deregistered scanner’s hook endpoint. The scanner SHOULD allow a brief grace period (implementation-defined, but typically a few seconds) for in-flight requests before considering the deregistration fully complete.

4.5.2. Scanner-Initiated Termination

The scanner MAY terminate a registration by setting its status to "deregistered" and returning an error to subsequent MTA requests against the registration. Implementation-specific policies govern when scanners do this, for example after detecting that the MTA has stopped sending hook requests for an extended period. The scanner SHOULD log such terminations and the MTA SHOULD treat any 404 REGISTRATION_NOT_FOUND response on a previously-active registration

as a signal to re-register.

5. Hook Request

When processing reaches a stage for which scanners are registered, the MTA constructs a hook request and transmits it to each registered scanner's hook endpoint.

5.1. Request Structure

Hook requests are transmitted via HTTP POST to the hook endpoint URL returned in the registration response. The Content-Type header indicates the serialization format negotiated during registration.

The following request headers apply to hook invocations:

X-MTA-Hooks-Registration
: REQUIRED. The registration identifier returned during registration. The scanner uses this to correlate the request with negotiated capabilities and to multiplex registrations sharing a hook endpoint host.

X-MTA-Hooks-Request-Id
: REQUIRED. A unique identifier for this logical hook invocation. The MTA MUST keep this value stable across retries of the same invocation. Scanners SHOULD deduplicate on this identifier to remain idempotent under retry. The value MUST be a string of at most 128 ASCII characters drawn from the unreserved set defined in [RFC3986].

Authorization (or transport-layer credentials)
: REQUIRED. See Section 5.2.

The request body contains a JSON or CBOR object with properties determined by the registration. The following sections describe available properties organized by category.

5.2. Authentication

The MTA MUST authenticate every hook invocation using the credentials provisioned for the registration. Scanners MUST verify the credentials and reject any unauthenticated hook request with HTTP 401. Scanners SHOULD also verify that the credentials are bound to the registration identified by X-MTA-Hooks-Registration; a mismatch MUST be rejected with HTTP 403.

The same authentication mechanisms supported for registration (Section 4.3) apply to hook invocations.

5.3. Common Properties

The following properties are available for both inbound and outbound processing:

stage: String
: The current processing stage.

action: String
: The action the MTA will perform if no scanner modifications occur. For inbound processing, values are "accept", "reject", "discard", "quarantine", or "disconnect". For outbound processing, values are "continue" or "cancel".

envelope: Envelope
: The message envelope containing sender and recipient information.

message: Object|null
: A structured representation of the email message based on the Email object defined in Section 4 of [RFC8621].

rawMessage: String|null
: The complete message in Internet Message Format as defined in [RFC5322]. In JSON serialization, the value is Base64-encoded. In CBOR serialization, the value is a raw byte string.

timestamp: UTCDate
: Timestamp when the current stage began.

protocol: ProtocolInfo|null
: Protocol information.

queue: QueueInfo|null
: Queue information.

server: ServerInfo|null
: Information about the MTA.

5.4. Envelope Properties

An *Envelope* object has the following properties:

from: EnvelopeAddress
: Sender information from MAIL FROM command.

to: EnvelopeAddress[]|DeliveryRecipient[]

Recipient information from RCPT TO commands. For inbound processing, this is an array of EnvelopeAddress objects. For outbound processing, this is an array of DeliveryRecipient objects.

An *EnvelopeAddress* object represents an address in the message envelope and has the following properties:

address: String|null The email address. For MAIL FROM, this MAY be null to represent the null reverse-path. For RCPT TO, this MUST NOT be null.

parameters: String[String] SMTP parameters associated with the address as key-value pairs. Common parameters include ORCPT, ENVID, and other DSN-related parameters defined in [RFC3461].

For outbound processing, recipient objects include additional delivery status fields described in Section 5.10.

5.5. Queue Properties

A *QueueInfo* object has the following properties:

id: String
Unique queue identifier for this message within this MTA. The identifier MUST be unique for the lifetime of the message in the queue and SHOULD remain stable across hook invocations for the same queued message.

5.6. Message Properties

5.6.1. Structured Message

The message property contains a structured representation of the email message based on the Email object defined in Section 4 of [RFC8621]. The following differences from the JMAP Email object apply:

* Metadata properties defined in Section 4.1.1 of [RFC8621] are not included, with the exception of the "size" property which MAY be present.

* The blobId property is replaced by a blob property containing the actual content. In JSON serialization, blob values are Base64-encoded strings. In CBOR serialization, blob values are raw byte strings.

Property availability depends on the processing stage and MTA capabilities. At early stages such as "connect" or "ehlo", message properties are not available.

5.6.2. Raw Message

The rawMessage property contains the complete message in Internet Message Format as defined in [RFC5322], including all MIME parts and attachments. In JSON serialization, the value is Base64-encoded. In CBOR serialization, the value is a raw byte string.

The rawMessage property represents the entire message as stored or received by the MTA. When present, it provides byte-exact access to the message content, which is necessary for operations such as cryptographic signature verification.

Scanners MAY request both message and rawMessage properties. If a scanner’s response modifies both properties, only rawMessage modifications are applied; message modifications are ignored.

The raw message content MUST NOT appear within the message property as a blob value. The message property contains only the structured parsed representation with individual body part contents available via bodyValues, while rawMessage contains the complete unparsed message.

5.6.3. Choosing Between Structured and Raw

The message property provides a structured, parsed representation suitable for:

* Header inspection and modification
* Recipient analysis
* Content-type aware body part processing
* Efficient access to specific message components

The rawMessage property provides the complete RFC 5322 message suitable for:

* Cryptographic verification (DKIM signature validation)
* Archival or compliance logging
* Processing that requires byte-exact message content

* Forwarding or re-injection without modification

Scanners that need both structured access and raw content SHOULD request both properties. When modifying messages, scanners SHOULD prefer structured modifications via the message property unless byte-exact control is required. Modifications to rawMessage require the scanner to produce a complete, valid RFC 5322 message.

5.7. Server Properties

A *ServerInfo* object has the following properties:

name: String
Server hostname.

ip: String
Server IP address.

port: UnsignedInt
Server port number.

5.8. Protocol Properties

A *ProtocolInfo* object has the following properties:

version: String
MTA Hooks protocol version.

5.9. Inbound Properties

The following properties are available only during inbound processing:

response: SmtpResponse|null The SMTP response the MTA would send if no scanner modifications occur.

client: ClientInfo|null Information about the connecting SMTP client.

senderAuth: SenderAuthentication|null Results of sender authentication checks.

auth: SmtpAuthentication|null SMTP authentication information, if the client authenticated.

tls: TlsInfo|null TLS connection information.

5.9.1. SMTP Response Properties

An *SmtpResponse* object represents an SMTP response and has the following properties:

code: UnsignedInt
The three-digit SMTP status code.

enhancedCode: String|null
The enhanced mail system status code as defined in [RFC3463], if available.

message: String
The human-readable response text.

5.9.2. Client Properties

A *ClientInfo* object has the following properties:

ip: String
Client IP address.

port: UnsignedInt
Client port number.

ptr: String|null
PTR record for client IP, if available.

ehlo: String
EHLO or HELO string sent by client.

asn: UnsignedInt|null
Autonomous System Number for client IP, if available.

country: String|null
ISO 3166-1 alpha-2 country code for client IP, if available.

activeConnections: UnsignedInt|null
Number of concurrent connections from this client.

5.9.3. TLS Properties

A *TlsInfo* object has the following properties:

version: String
TLS protocol version (e.g., "TLSv1.3").

cipher: String

Cipher suite name.

cipherBits: UnsignedInt
Cipher key length in bits.

certIssuer: String|null
Client certificate issuer, if a client certificate was presented.

certSubject: String|null
Client certificate subject, if a client certificate was presented.

5.9.4. Sender Authentication Properties

A *SenderAuthentication* object has the following properties:

spf-ehlo: String|null SPF result for EHLO identity as defined in [RFC7208].

spf-mail: String|null SPF result for MAIL FROM identity as defined in [RFC7208].

dkim: String|null DKIM verification result as defined in [RFC6376].

arc: String|null ARC verification result.

dmarc: String|null DMARC evaluation result as defined in [RFC7489].

Authentication result values follow the conventions established in the respective specifications.

5.9.5. SMTP Authentication Properties

An *SmtpAuthentication* object has the following properties:

login: String
Authenticated username.

method: String
SASL mechanism used for authentication.

5.10. Outbound Properties

The following properties are available only during outbound processing.

5.10.1. Extended Queue Properties

For outbound processing, the queue property uses an *OutboundQueueInfo* object, which extends QueueInfo with additional fields:

id: String
: Unique queue identifier for this message.

expiresAt: UTCDate|null
: Timestamp when the queue entry expires.

attempts: UnsignedInt
: Total delivery attempts made so far.

5.10.2. Recipient Delivery Status

For outbound processing, each recipient in envelope.to is represented as a DeliveryRecipient object.

A *DeliveryRecipient* object extends EnvelopeAddress with delivery status information and has the following properties:

address: String
: The recipient email address.

parameters: String[String]
: SMTP parameters associated with the address as key-value pairs.

status: String
: The delivery status for this recipient. Values are:

  * "pending": The MTA has not yet attempted delivery for this recipient.
  * "delivered": Delivery succeeded.
  * "deferred": Delivery failed transiently and will be retried per the schedule in nextAttemptAt.
  * "failed": Delivery failed permanently and the MTA will generate a failure DSN.
  * "failed-silent": Delivery is suppressed and the MTA MUST NOT generate any DSN for this recipient. Typically set by a scanner via modification to drop a recipient without notifying the sender.

attempt: UnsignedInt
The current delivery attempt number for this recipient.

lastResponse: SmtpResponse|null
The last SMTP response received for this recipient, or null if no attempt has been made.

nextAttemptAt: UTCDate|null
The scheduled time for the next delivery attempt, or null if no retry is scheduled.

nextDsnAt: UTCDate|null
The scheduled time for the next DSN generation, or null if no DSN is scheduled.

queueName: String|null
The name of the outbound queue handling this recipient, if applicable.

For the "dsn" stage, the message and rawMessage properties contain the DSN message that the MTA is about to send. Scanners MAY modify or delete these properties to alter or suppress DSN generation.

5.11. Example Inbound Hook Request

```
POST /v1/hooks/invoke/reg_7a3b9c2e-4d5f-6a7b-8c9d-0e1f2a3b4c5d \
  HTTP/1.1
Host: scanner.example.com
Content-Type: application/json
Authorization: Bearer eyJhbGciOiJIUzI1NiIs...
X-MTA-Hooks-Registration: reg_7a3b9c2e-4d5f-6a7b-8c9d-0e1f2a3b4c5d
X-MTA-Hooks-Request-Id: 01HZ8K3X7F4Y5N6Q9R2S3T4U5V

{
  "stage": "data",
  "action": "accept",
  "timestamp": "2024-12-21T16:30:00Z",

  "response": {
    "code": 250,
    "enhancedCode": "2.0.0",
    "message": "OK"
  },

  "protocol": {
    "version": "1.0"
  },
```

```
    "queue": {
      "id": "queue_abc123"
    },

    "envelope": {
      "from": {
        "address": "sender@example.org",
        "parameters": {}
      },
      "to": [
        {
          "address": "recipient@example.com",
          "parameters": {}
        }
      ]
    },

    "message": {
      "size": 2048,
      "subject": "Meeting Tomorrow",
      "from": [
        {
          "name": "Sender Name",
          "email": "sender@example.org"
        }
      ],
      "to": [
        {
          "email": "recipient@example.com"
        }
      ],
      "sentAt": "2024-12-21T16:29:00Z",
      "messageId": ["<msg123@example.org>"],
      "headers": [
        {"name": "From", "value": "Sender Name <sender@example.org>"},
        {"name": "To", "value": "recipient@example.com"},
        {"name": "Subject", "value": "Meeting Tomorrow"},
        {"name": "Date", "value": "Sat, 21 Dec 2024 16:29:00 +0000"},
        {"name": "Message-ID", "value": "<msg123@example.org>"}
      ],
      "bodyStructure": {
        "type": "text/plain",
        "charset": "utf-8",
        "size": 28
      },
      "bodyValues": {
        "1": {
          "value": "Let's meet tomorrow at 10am.",
```

```
            "isEncodingProblem": false,
            "isTruncated": false
          }
        }
      },

      "client": {
        "ip": "192.0.2.100",
        "port": 54321,
        "ptr": "mail.example.org",
        "ehlo": "mail.example.org",
        "activeConnections": 3
      },

      "server": {
        "name": "mx1.example.com",
        "ip": "198.51.100.25",
        "port": 25
      },

      "tls": {
        "version": "TLSv1.3",
        "cipher": "TLS_AES_256_GCM_SHA384",
        "cipherBits": 256
      },

      "senderAuth": {
        "spf-mail": "pass",
        "dkim": "pass",
        "dmarc": "pass"
      }
    }
```

5.12. Example Outbound Hook Request

```
   POST /v1/hooks/invoke/reg_7a3b9c2e-4d5f-6a7b-8c9d-0e1f2a3b4c5d \
     HTTP/1.1
   Host: scanner.example.com
   Content-Type: application/json
   Authorization: Bearer eyJhbGciOiJIUzI1NiIs...
   X-MTA-Hooks-Registration: reg_7a3b9c2e-4d5f-6a7b-8c9d-0e1f2a3b4c5d
   X-MTA-Hooks-Request-Id: 01HZ8K9P2QABCDEFGH7J8K9L0M

   {
     "stage": "delivery",
     "action": "continue",
     "timestamp": "2024-12-21T17:00:00Z",
```

```
    "protocol": {
      "version": "1.0"
    },

    "queue": {
      "id": "queue_def456",
      "expiresAt": "2024-12-24T17:00:00Z",
      "attempts": 3
    },

    "envelope": {
      "from": {
        "address": "notifications@example.com",
        "parameters": {}
      },
      "to": [
        {
          "address": "user1@recipient.example",
          "parameters": {},
          "status": "delivered",
          "attempt": 1,
          "lastResponse": {
            "code": 250,
            "enhancedCode": "2.0.0",
            "message": "Message accepted"
          }
        },
        {
          "address": "user2@recipient.example",
          "parameters": {},
          "status": "deferred",
          "attempt": 3,
          "lastResponse": {
            "code": 451,
            "enhancedCode": "4.7.1",
            "message": "Try again later"
          },
          "nextAttemptAt": "2024-12-21T18:00:00Z",
          "nextDsnAt": "2024-12-22T17:00:00Z"
        }
      ]
    },

    "message": {
      "subject": "Your order has shipped",
      "from": [{"email": "notifications@example.com"}],
      "to": [
        {"email": "user1@recipient.example"},
```

```
        {"email": "user2@recipient.example"}
      ],
      "size": 4096
    },

    "server": {
      "name": "smtp-out.example.com",
      "ip": "198.51.100.50",
      "port": 25
    }
  }
```

6. Hook Response

Scanners respond to hook requests with modifications to the request object. These modifications instruct the MTA how to proceed with message processing.

6.1. Response Structure

On success, scanners respond with HTTP 200 OK and a body conforming to this section, or with HTTP 204 No Content and an empty body to indicate "no modifications". The response body contains a JSON or CBOR object specifying modifications to apply to the hook request. The object has the following fields:

set: SetOperation[]|null An array of set operations to apply, or null if no set operations.

add: AddOperation[]|null An array of add operations to apply, or null if no add operations.

delete: DeleteOperation[]|null An array of delete operations to apply, or null if no delete operations.

A *SetOperation* object has the following properties:

path: String
JSON Pointer (per [RFC6901]) to the property to replace.

value: any
The new value for the property.

An *AddOperation* object has the following properties:

path: String
JSON Pointer to the location where the value should be added.

value: any
The value to add.

index: UnsignedInt|null
For array additions, the zero-based index at which to insert the value. If null or omitted, the value is appended to the array.

A *DeleteOperation* object has the following properties:

path: String
JSON Pointer to the property to remove.

6.2. Actions

Scanners change the MTA's action by including a set operation targeting the "/action" path. The following values are valid for inbound and outbound processing respectively.

The following action values are valid:

6.2.1. Inbound Actions

accept
Accept the message for delivery to recipients. Non-terminal; chain continues.

reject
Reject the message and return an error response to the sending client. Terminal; chain stops.

discard
Accept the message from the client but do not deliver it. The client receives a success response. Terminal; chain stops.

quarantine
Accept the message and place it in quarantine for administrative review. Quarantine location and handling are implementation-defined. Non-terminal; chain continues.

disconnect
Terminate the SMTP connection immediately. Terminal; chain stops.

6.2.2. Outbound Actions

continue Proceed with normal processing. Non-terminal; chain continues.

cancel Cancel all pending deliveries for this message. Terminal;

chain stops.

6.3. Modifications

Scanners communicate changes by specifying operations on the hook request object. The response contains JSON Pointer paths identifying properties to modify and the operations to perform. These modifications can alter any aspect of the transaction that the MTA permits, including:

* The action the MTA should take (accept, reject, quarantine, etc.)

* The SMTP response to send to the client or expect from the server

* Message headers and body content

* Envelope addresses (sender and recipients)

* Per-recipient delivery status for outbound processing

Three modification operations are supported:

set
: Replace the value at a specified path with a new value.

add
: Insert a value at a specified path. For objects, this adds a new property. For arrays, this inserts an element at the specified index or appends if no index is given.

delete
: Remove the value at a specified path.

6.3.1. Set Operations

Set operations replace values at specified paths.

```
{
  "set": [
    {
      "path": "/action",
      "value": "reject"
    },
    {
      "path": "/response",
      "value": {
        "code": 550,
        "enhancedCode": "5.7.1",
        "message": "Message rejected due to policy"
      }
    }
  ]
}
```

6.3.2. Add Operations

Add operations insert new values. For arrays, an optional index specifies insertion position.

```
{
  "add": [
    {
      "path": "/message/headers",
      "value": {"name": "X-Spam-Score", "value": "5.2"},
      "index": 0
    },
    {
      "path": "/envelope/to",
      "value": {
        "address": "archive@example.com",
        "parameters": {}
      }
    }
  ]
}
```

6.3.3. Delete Operations

Delete operations remove values at specified paths. When deleting array elements, indices refer to positions in the array at the time each delete operation executes. Because delete operations execute sequentially and earlier deletions cause subsequent elements to shift to lower indices, scanners SHOULD order array deletions from highest to lowest index to avoid unexpected results.

For example, to delete elements originally at indices 1 and 3 from an array:

```
{
  "delete": [
    {"path": "/message/headers/3"},
    {"path": "/message/headers/1"}
  ]
}
```

Deleting index 3 first leaves the element originally at index 1 still at index 1. If the order were reversed, deleting index 1 first would shift the element originally at index 3 to index 2.

6.3.4. Modification Order

The MTA applies modifications in the following order:

1. Set operations
2. Add operations
3. Delete operations

This ordering allows predictable results when multiple operations affect related paths.

To change the action the MTA takes, the scanner sets the "/action" path to the desired value. To modify the SMTP response, the scanner can either set individual response fields (e.g., "/response/code", "/response/message") or replace the entire response object by setting "/response".

If a scanner modifies both the "message" and "rawMessage" properties, the "rawMessage" modification takes precedence and "message" modifications are ignored.

6.3.5. Conflicting Operations

If a response contains multiple operations targeting the same path or overlapping paths, the MTA applies them in the defined order (set, then add, then delete). The final state reflects all operations applied sequentially. For example, if a response sets "/message/subject" and also deletes "/message/subject", the subject will be deleted (delete operations execute last).

Implementations SHOULD NOT submit responses with conflicting operations targeting the same path, as the resulting behavior, while deterministic, may be confusing.

6.3.6. No Modification Response

If a scanner has no modifications to request, it MUST return either:

* An empty JSON/CBOR object: {}
* A response with empty or null modification arrays: {"set": null, "add": null, "delete": null}
* An HTTP 204 No Content response with no body

In all cases, the MTA proceeds with the action specified in the original request. The scanner chain continues to the next registered scanner unless the current action is terminal.

6.3.7. Size Limits

MTAs MAY impose limits on the size of modification values. If a modification would cause a header or message to exceed configured size limits, the MTA SHOULD reject that specific modification and log the failure.

6.3.8. Modification Errors

If a modification cannot be applied (for example, the path references a non-existent property for deletion, or the path is not in the permitted updateProperties list), the MTA SHOULD:

1. Log the error with sufficient detail for debugging
2. Increment modification failure statistics
3. Continue processing remaining modifications in the response

The MTA MUST NOT reject the entire response due to a single failed modification unless the failed modification is critical to interpreting the response (such as an invalid action value).

6.4. Error Responses

A scanner that cannot process a hook request returns an HTTP 4xx or 5xx status with a body that follows the error structure defined in Appendix A. Status code semantics:

* 401 Unauthorized: The MTA's credentials are missing or invalid. The MTA MUST NOT retry without acquiring fresh credentials.

* 403 Forbidden: The credentials are valid but not authorized for this registration or operation. The MTA MUST NOT retry without operator intervention.

* 404 Not Found: The registration referenced by X-MTA-Hooks-Registration is unknown to the scanner. The MTA SHOULD treat the registration as terminated and re-register before sending further hooks.

* 410 Gone: The registration has been deregistered. The MTA MUST stop sending hooks under this registration; the MTA MAY re-register.

* 413 Content Too Large: The request body exceeds the scanner's maxMessageSize. The MTA MUST NOT retry the same payload.

* 422 Unprocessable Content: The request was syntactically valid but referenced unsupported properties or values. Treated as a 4xx; do not retry.

* 429 Too Many Requests: The scanner is rate-limiting. The MTA SHOULD honor the Retry-After header if present.

* 5xx: A transient scanner error. The MTA MAY retry per its retry policy (Section 7.5.3).

If the response body is not parseable as the negotiated serialization, or if it conforms to the response schema but contains modifications targeting paths outside the registration's negotiated updateProperties, the MTA MUST treat the response as malformed: it MUST NOT apply any modifications from the response, MUST proceed with the default action carried in the request, MUST log the failure with the request identifier, and MAY count the response toward the scanner's failure statistics for circuit-breaking purposes.

6.5. Example Hook Responses

6.5.1. Accept with Header Addition

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "add": [
    {
      "path": "/message/headers",
      "value": {"name": "X-Spam-Status", "value": "No"},
      "index": 0
    },
    {
      "path": "/message/headers",
      "value": {"name": "X-Spam-Score", "value": "1.2"},
      "index": 1
    }
  ]
}
```

6.5.2. Reject with Custom Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "set": [
    {
      "path": "/action",
      "value": "reject"
    },
    {
      "path": "/response",
      "value": {
        "code": 550,
        "enhancedCode": "5.7.1",
        "message": "Message rejected: virus detected"
      }
    }
  ]
}
```

6.5.3. Modify Subject and Add Recipient

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "set": [
    {
      "path": "/message/subject",
      "value": "[EXTERNAL] Original Subject"
    }
  ],
  "add": [
    {
      "path": "/envelope/to",
      "value": {
        "address": "compliance@example.com",
        "parameters": {}
      }
    }
  ]
}
```

6.5.4. Cancel Specific Recipient Delivery

For outbound processing, to cancel delivery for a specific recipient:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "set": [
    {
      "path": "/envelope/to/1/status",
      "value": "failed-silent"
    }
  ]
}
```

6.5.5. No Action Response

If a scanner has no changes to request, it returns an empty response body or a response with no action or modifications:

```
HTTP/1.1 200 OK
Content-Type: application/json

{}
```

The MTA proceeds with the default action specified in the request.

7. Transport

This section specifies HTTP transport requirements for MTA Hooks communications.

7.1. HTTP Version

Implementations MUST support HTTP/1.1 [RFC9112]. Implementations SHOULD support HTTP/2 [RFC9113] for improved performance through multiplexing. Implementations MAY support HTTP/3 [RFC9114].

When multiple HTTP versions are available, implementations SHOULD prefer newer versions for their performance benefits while maintaining fallback to HTTP/1.1.

7.2. TLS Requirements

All MTA Hooks communications MUST use TLS. Implementations MUST support TLS 1.2 [RFC5246] and SHOULD support TLS 1.3 [RFC8446].

Implementations MUST validate server certificates against trusted certificate authorities. Self-signed certificates SHOULD NOT be accepted in production deployments unless explicitly configured by administrators.

Cipher suite selection SHOULD follow current best practices. Implementations SHOULD disable cipher suites known to be weak or compromised.

7.3. Request Methods

The following HTTP methods are used:

* GET: Discovery document retrieval and status queries

* POST: Registration requests and hook invocations

* DELETE: Deregistration requests

7.4. Content Negotiation

For hook requests and responses, the Content-Type header MUST be set to "application/json" for JSON serialization or "application/cbor" for CBOR serialization.

The Accept header MAY be used during discovery to indicate preferred serialization format. If the scanner cannot provide the requested format, it SHOULD return the discovery document in JSON format.

7.5. Error Handling

7.5.1. HTTP Status Codes

MTAs SHOULD interpret HTTP status codes as follows:

* 2xx: Request successful. Process response body.
* 4xx: Client error. Do not retry; log error and proceed with default action.
* 5xx: Server error. May retry according to policy.

7.5.2. Timeout Handling

If a scanner does not respond within the configured timeout, the MTA SHOULD proceed with the default action. Timeout handling policy (fail-open vs. fail-closed) is implementation-defined and SHOULD be configurable.

7.5.3. Retry Policy

For transient errors (5xx status codes, network failures, timeouts), implementations SHOULD implement retry with exponential backoff. A recommended policy is:

* Maximum 3 retry attempts
* Initial delay: 100 milliseconds
* Backoff multiplier: 2
* Maximum delay: 5 seconds
* Add random jitter of 0-100 milliseconds

The MTA MUST keep the X-MTA-Hooks-Request-Id header value stable across retries of the same logical invocation, so scanners can deduplicate. Implementations MAY provide configuration options to adjust retry parameters.

7.6. Connection Management

Implementations SHOULD use connection pooling to reduce latency for repeated requests to the same scanner endpoint. HTTP/2 multiplexing, where available, reduces the need for multiple connections.

Implementations SHOULD respect HTTP keep-alive semantics and connection limits advertised by scanners.

8. Implementation Considerations

8.1. MTA Considerations

8.1.1. Unavailable Scanners

When a scanner becomes unavailable (timeouts, errors, deregistration), the MTA must determine how to proceed. Common policies include:

* Fail-open: Proceed with default action as if scanner approved

* Fail-closed: Reject or defer message processing

The appropriate policy depends on deployment requirements. Security-focused deployments may prefer fail-closed, while availability-focused deployments may prefer fail-open. Implementations SHOULD make this policy configurable.

8.1.2. Performance Considerations

Synchronous hook invocation adds latency to mail processing. Implementations SHOULD consider:

* Connection pooling and keep-alive for scanner connections

* Parallel invocation of independent scanners where ordering is not required

* Timeout tuning based on scanner response time characteristics

8.2. Scanner Considerations

8.2.1. Idempotency

Scanners SHOULD design their processing to be idempotent. Network issues or MTA retries may result in duplicate invocations for the same message. Scanners SHOULD handle duplicate requests gracefully.

8.2.2. Response Time

Scanners operate in the critical path of mail delivery. Long response times delay message processing and may cause timeouts. Scanners SHOULD:

* Process requests promptly

* Implement internal timeouts shorter than MTA timeouts

* Consider asynchronous processing for expensive operations

* Return default responses when processing cannot complete in time

8.2.3. Stateless Design

Scanners SHOULD avoid relying on state from previous invocations. Each hook request contains complete context for processing decisions. Stateless design improves reliability and simplifies horizontal scaling.

8.3. Large Message Handling

Large messages present challenges for both MTAs and scanners. Implementations SHOULD consider:

* Message size limits advertised in discovery documents

* Truncation policies for oversized messages

* Memory management for large message bodies

Streaming or chunked delivery is not currently specified. For very large messages, implementations MAY arrange out-of-band content retrieval, though this is outside the scope of this specification.

8.4. High Availability

8.4.1. Scanner High Availability

For production deployments, scanners SHOULD be deployed with redundancy. Approaches include:

* Multiple scanner instances behind a load balancer

* Health checking with automatic failover

* Geographic distribution for disaster recovery

Hook endpoints typically resolve to a load-balanced address; the MTA holds a single endpoint URL per registration.

8.4.2. Registration State

Registration state resides at the scanner. Scanner instances behind a load balancer MUST share registration state (or partition deterministically by registration identifier) so that hook requests can be served regardless of which instance receives them. The MTA SHOULD treat 404 REGISTRATION_NOT_FOUND on a previously-active registration as a signal to re-register rather than as a permanent failure.

9. Security Considerations

9.1. Trust Model

MTA Hooks places the MTA exclusively in the role of HTTP client. The MTA does not expose any HTTP server to scanners and does not accept inbound registration requests. This containment is deliberate: it eliminates a category of attacks (registration flood, server-side request forgery via callback verification, exposure of MTA management endpoints) that would otherwise threaten the most security-critical component on the mail path.

The trust boundary runs in one direction. The scanner operator decides which MTAs may register and provisions credentials accordingly. The MTA admin decides which scanners to call out to and configures them out of band. Mutual trust is established before any protocol traffic flows.

9.2. Authentication and Authorization

Authentication of every MTA-to-scanner request (registration, status, deregistration, hook invocation) is REQUIRED. Scanners MUST reject unauthenticated requests with HTTP 401. The specific authentication mechanism is out of scope; recommended approaches:

* Bearer tokens: Simple to implement; tokens MUST be generated with sufficient entropy (at least 128 bits) and rotated periodically. Tokens SHOULD be scoped to a single MTA identity.

* Mutual TLS: Provides strong authentication and binds identity to transport security. RECOMMENDED for high-trust deployments.

* HMAC signatures: Allows authentication without transmitting bearer secrets; requires secure key distribution.

Authorization policies at the scanner SHOULD restrict, per-MTA-identity, which stages may be subscribed to, which properties may be requested, and which actions may be performed. Policies SHOULD be

enforced both at registration time (rejecting impermissible registrations) and at hook-response time (ignoring out-of-policy modifications).

9.3. Credential Provisioning

Scanner operators issue credentials to MTA operators through an out-of-band channel that this specification does not define. Credential provisioning practices SHOULD include:

* Per-MTA-instance credentials (avoid shared secrets across multiple MTAs).

* Credential rotation on a defined cadence.

* Immediate revocation when an MTA instance is decommissioned or compromised.

* Storage of credentials at the MTA in protected configuration that is not logged or exported with diagnostics.

Scanner-issued credentials SHOULD encode the MTA identity in a way that survives renewal of the underlying secret (for example, a stable client-certificate Subject CN, or a tenant identifier conveyed alongside a rotated bearer token).

9.4. Transport Security

All MTA Hooks communications MUST use TLS to protect message content and authentication credentials in transit. Implementations MUST validate certificates to prevent man-in-the-middle attacks.

For internal network deployments, administrators may choose to use private certificate authorities. However, disabling certificate validation is NOT RECOMMENDED even for internal communications.

9.5. Message Confidentiality

Email messages frequently contain sensitive personal or business information. Scanners necessarily receive access to message content to perform their functions. Deployments SHOULD consider:

* Data handling policies for scanner operators

* Logging policies that avoid recording message content

* Data retention limits at scanner endpoints

* Regulatory requirements such as GDPR for personal data processing

9.6. Denial of Service

9.6.1. Registration Flood

Although the registration endpoint is on the scanner, an attacker that obtains valid MTA credentials could flood the scanner with registration churn. Scanners SHOULD:

* Authenticate before parsing any registration body.

* Rate limit registration and deregistration per MTA identity.

* Limit maximum concurrent registrations per MTA identity.

* Apply the same protections to the discovery endpoint, which is unauthenticated by design.

9.6.2. Scanner Resource Exhaustion

A compromised or misconfigured MTA could overwhelm a scanner with hook requests. Scanners SHOULD:

* Implement per-registration rate limiting and concurrency caps.

* Set appropriate request body size limits, advertised in the discovery document.

* Monitor for unusual invocation patterns and suspend offending registrations.

9.6.3. Slow Scanner Attacks

Slow scanner responses can exhaust MTA resources (connection slots, memory, queued mail). MTAs SHOULD:

* Enforce strict timeouts; never wait indefinitely for a hook response.

* Limit concurrent requests to any single scanner.

* Implement circuit breaker patterns for repeatedly slow or failing scanners.

9.6.4. Compromised Scanner

A compromised scanner can attempt to coerce the MTA into mishandling mail by returning malicious modifications. The MTA's defense in depth is to enforce its declared updateProperties strictly, validate every modification (Section 9.7), and reject responses that violate the negotiated capabilities. The MTA SHOULD NOT depend on the scanner to behave honestly.

9.7. Injection Attacks

Scanner responses contain modifications applied to message processing. Implementations MUST validate all scanner-provided data:

* Header names and values MUST be validated to prevent header injection (CR/LF smuggling, malformed encoded-words, header folding abuse).

* Recipient addresses MUST be syntactically validated and re-checked against MTA policy before being added to the envelope.

* Body modifications MUST not introduce malformed MIME structure or content that violates declared character sets.

* Modifications targeting paths outside the registration's negotiated updateProperties MUST be discarded.

9.8. Privacy Considerations

Email processing inherently involves access to personal communications. Implementations SHOULD minimize data exposure:

* Scanners SHOULD request only necessary properties

* MTAs SHOULD honor property restrictions from capability negotiation

* Logging SHOULD avoid recording message content or metadata

* Scanner operators SHOULD implement appropriate data protection measures

10. IANA Considerations

10.1. Well-Known URI Registration

This document registers the following well-known URI per [RFC8615]:

URI suffix: mta-hooks

Change controller: IETF

Specification document: This document

Status: permanent

Related information: N/A

10.2. MTA Hooks Serialization Format Registry

IANA is requested to create a new registry entitled "MTA Hooks Serialization Formats" with the following initial contents:

* Format: json
  - Description: JSON serialization per RFC 8259
  - Reference: This document
* Format: cbor
  - Description: CBOR serialization per RFC 8949
  - Reference: This document

New registrations require Specification Required per [RFC8126].

10.3. MTA Hooks Inbound Action Registry

IANA is requested to create a new registry entitled "MTA Hooks Inbound Actions" with the following initial contents:

* Action: accept
  - Description: Accept message for delivery
  - Reference: This document
* Action: reject
  - Description: Reject message with error response
  - Reference: This document
* Action: discard

- Description: Accept but do not deliver message
- Reference: This document

* Action: quarantine
  - Description: Place message in quarantine
  - Reference: This document
* Action: disconnect
  - Description: Terminate SMTP connection
  - Reference: This document

New registrations require Specification Required per [RFC8126].

10.4. MTA Hooks Outbound Action Registry

IANA is requested to create a new registry entitled "MTA Hooks Outbound Actions" with the following initial contents:

* Action: continue
  - Description: Proceed with normal delivery processing
  - Reference: This document
* Action: cancel
  - Description: Cancel pending deliveries
  - Reference: This document

New registrations require Specification Required per [RFC8126].

10.5. MTA Hooks Inbound Stage Registry

IANA is requested to create a new registry entitled "MTA Hooks Inbound Stages" with the following initial contents:

* Stage: connect
  - Description: Client connection established
  - Reference: This document

* Stage: ehlo
  - Description: EHLO/HELO command received
  - Reference: This document
* Stage: mail
  - Description: MAIL FROM command received
  - Reference: This document
* Stage: rcpt
  - Description: RCPT TO command received
  - Reference: This document
* Stage: data
  - Description: Message content received
  - Reference: This document

New registrations require Specification Required per [RFC8126].

10.6. MTA Hooks Outbound Stage Registry

IANA is requested to create a new registry entitled "MTA Hooks Outbound Stages" with the following initial contents:

* Stage: delivery
  - Description: Delivery attempt completed
  - Reference: This document
* Stage: defer
  - Description: Delivery deferred for retry
  - Reference: This document
* Stage: dsn
  - Description: DSN generation pending
  - Reference: This document

New registrations require Specification Required per [RFC8126].

10.7. MTA Hooks Error Code Registry

IANA is requested to create a new registry entitled "MTA Hooks Error Codes" with the initial contents specified in Appendix A.

New registrations require Specification Required per [RFC8126].

11. References

11.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002, <https://www.rfc-editor.org/rfc/rfc3339>.

[RFC3463] Vaudreuil, G., "Enhanced Mail System Status Codes", RFC 3463, DOI 10.17487/RFC3463, January 2003, <https://www.rfc-editor.org/rfc/rfc3463>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/rfc/rfc3986>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/rfc/rfc5246>.

[RFC5321] Klensin, J., "Simple Mail Transfer Protocol", RFC 5321, DOI 10.17487/RFC5321, October 2008, <https://www.rfc-editor.org/rfc/rfc5321>.

[RFC5322] Resnick, P., Ed., "Internet Message Format", RFC 5322, DOI 10.17487/RFC5322, October 2008, <https://www.rfc-editor.org/rfc/rfc5322>.

[RFC6901] Bryan, P., Ed., Zyp, K., and M. Nottingham, Ed., "JavaScript Object Notation (JSON) Pointer", RFC 6901, DOI 10.17487/RFC6901, April 2013, <https://www.rfc-editor.org/rfc/rfc6901>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/rfc/rfc8126>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/rfc/rfc8259>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

[RFC8615] Nottingham, M., "Well-Known Uniform Resource Identifiers (URIs)", RFC 8615, DOI 10.17487/RFC8615, May 2019, <https://www.rfc-editor.org/rfc/rfc8615>.

[RFC8620] Jenkins, N. and C. Newman, "The JSON Meta Application Protocol (JMAP)", RFC 8620, DOI 10.17487/RFC8620, July 2019, <https://www.rfc-editor.org/rfc/rfc8620>.

[RFC8621] Jenkins, N. and C. Newman, "The JSON Meta Application Protocol (JMAP) for Mail", RFC 8621, DOI 10.17487/RFC8621, August 2019, <https://www.rfc-editor.org/rfc/rfc8621>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/rfc/rfc8949>.

[RFC9112] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP/1.1", STD 99, RFC 9112, DOI 10.17487/RFC9112, June 2022, <https://www.rfc-editor.org/rfc/rfc9112>.

[RFC9113] Thomson, M., Ed. and C. Benfield, Ed., "HTTP/2", RFC 9113, DOI 10.17487/RFC9113, June 2022, <https://www.rfc-editor.org/rfc/rfc9113>.

[RFC9114] Bishop, M., Ed., "HTTP/3", RFC 9114, DOI 10.17487/RFC9114, June 2022, <https://www.rfc-editor.org/rfc/rfc9114>.

11.2. Informative References

[RFC3461] Moore, K., "Simple Mail Transfer Protocol (SMTP) Service Extension for Delivery Status Notifications (DSNs)", RFC 3461, DOI 10.17487/RFC3461, January 2003, <https://www.rfc-editor.org/rfc/rfc3461>.

[RFC6376] Crocker, D., Ed., Hansen, T., Ed., and M. Kucherawy, Ed., "DomainKeys Identified Mail (DKIM) Signatures", STD 76, RFC 6376, DOI 10.17487/RFC6376, September 2011, <https://www.rfc-editor.org/rfc/rfc6376>.

[RFC7208] Kitterman, S., "Sender Policy Framework (SPF) for Authorizing Use of Domains in Email, Version 1", RFC 7208, DOI 10.17487/RFC7208, April 2014, <https://www.rfc-editor.org/rfc/rfc7208>.

[RFC7489] Kucherawy, M., Ed. and E. Zwicky, Ed., "Domain-based Message Authentication, Reporting, and Conformance (DMARC)", RFC 7489, DOI 10.17487/RFC7489, March 2015, <https://www.rfc-editor.org/rfc/rfc7489>.

Appendix A. Error Codes

This appendix defines error codes used in MTA Hooks error responses.

A.1. Error Response Structure

Error responses use the following structure:

```
{
  "error": {
    "code": "ERROR_CODE",
    "message": "Human-readable error description"
  }
}
```

A.2. HTTP 400 Bad Request

* INVALID_REQUEST: Malformed JSON or CBOR, or missing required fields.

* CAPABILITY_MISMATCH: Requested capability not supported by the scanner.

* INVALID_STAGE: One or more requested stages are not valid.

* INVALID_ACTION: One or more requested actions are not valid.

* INVALID_MODIFICATION: One or more requested modifications are not valid.

* NO_STAGES_REQUESTED: Neither inbound nor outbound stages were specified.

* FILTER_INVALID: Filter configuration is syntactically invalid.

* MISSING_REQUEST_ID: The X-MTA-Hooks-Request-Id header is missing on a hook invocation.

A.3. HTTP 401 Unauthorized

* AUTHENTICATION_REQUIRED: No authentication credentials provided.

* INVALID_CREDENTIALS: Provided credentials are invalid or expired.

A.4. HTTP 403 Forbidden

* REGISTRATION_DENIED: Valid credentials but the MTA identity is not authorized to register.

* DOMAIN_NOT_ALLOWED: MTA not authorized for the requested domains.

* REGISTRATION_MISMATCH: The credentials are not bound to the registration identified by X-MTA-Hooks-Registration.

A.5. HTTP 404 Not Found

* REGISTRATION_NOT_FOUND: No registration exists with the specified identifier.

A.6. HTTP 409 Conflict

* REGISTRATION_LIMIT_REACHED: Maximum number of registrations for this MTA identity exceeded.

A.7. HTTP 410 Gone

* REGISTRATION_DEREGISTERED: The registration has been deregistered and will not be re-activated.

A.8. HTTP 413 Content Too Large

* REQUEST_TOO_LARGE: The request body exceeds the scanner’s advertised maxMessageSize.

A.9. HTTP 422 Unprocessable Content

* UNSUPPORTED_FILTER: The filter contains conditions that are not supported by this scanner.

* UNSUPPORTED_PROPERTY: One or more requested properties are not in the scanner’s advertised fetchProperties.

A.10. HTTP 429 Too Many Requests

* RATE_LIMITED: Too many requests; retry after the time specified in the Retry-After header.

A.11. HTTP 500 Internal Server Error

* INTERNAL_ERROR: Unexpected server error during processing.

A.12. HTTP 503 Service Unavailable

* REGISTRATION_DISABLED: Registration temporarily disabled by administrator.

* SCANNER_UNAVAILABLE: The scanner is temporarily unable to process hook invocations.

Appendix B. Complete Example Flows

This appendix provides complete request and response examples for common scenarios.

B.1. Inbound Spam Filtering

This example shows a complete flow for spam filtering during inbound message reception.

B.1.1. Discovery

```
GET /.well-known/mta-hooks HTTP/1.1
Host: scanner.example.com
Accept: application/json
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "version": "1.0",
  "endpoints": {
    "registration": "/v1/hooks/register",
    "deregistration": "/v1/hooks/register/{registration_id}"
  },
  "serialization": ["json"],
  "capabilities": {
    "inbound": {
      "stages": ["data"],
      "actions": ["accept", "reject", "quarantine"],
      "fetchProperties": ["/message", "/envelope", "/senderAuth",
                          "/client"],
      "updateProperties": ["/message/headers", "/action", "/response"]
    }
  },
  "limits": {
    "maxMessageSize": 26214400,
    "timeoutMs": 30000
  }
}
```

B.1.2. Registration

```
POST /v1/hooks/register HTTP/1.1
Host: scanner.example.com
Content-Type: application/json
Authorization: Bearer sk_live_abc123

{
  "name": "Acme MTA mx1",
  "version": "3.0.1",
  "timeoutMs": 20000,
  "serialization": "json",
  "inbound": {
    "stages": ["data"],
    "properties": ["/message", "/envelope", "/senderAuth", "/client"]
  },
  "metadata": {
    "operator": "acme-corp"
  }
}
```

B.1.3. Registration Response

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: /v1/hooks/register/reg_spam_001

{
  "registrationId": "reg_spam_001",
  "status": "active",
  "createdAt": "2024-12-21T10:00:00Z",
  "expiresAt": "2025-01-21T10:00:00Z",
  "hookEndpoint": "/v1/hooks/invoke/reg_spam_001",
  "negotiated": {
    "serialization": "json",
    "inbound": {
      "stages": ["data"],
      "properties": ["/message", "/envelope", "/senderAuth", "/client"]
    }
  },
  "endpoints": {
    "deregistration": "/v1/hooks/register/reg_spam_001",
    "status": "/v1/hooks/register/reg_spam_001/status"
  }
}
```

B.1.4. Hook Invocation - Clean Message

```
POST /v1/hooks/invoke/reg_spam_001 HTTP/1.1
Host: scanner.example.com
Content-Type: application/json
Authorization: Bearer sk_live_abc123
X-MTA-Hooks-Registration: reg_spam_001
X-MTA-Hooks-Request-Id: 01HZ8K3X7F4Y5N6Q9R2S3T4U5V

{
  "stage": "data",
  "action": "accept",
  "timestamp": "2024-12-21T10:30:00Z",
  "response": {
    "code": 250,
    "enhancedCode": "2.0.0",
    "message": "OK"
  },
  "envelope": {
    "from": {"address": "alice@sender.example", "parameters": {}},
    "to": [{"address": "bob@recipient.example", "parameters": {}}]
  },
  "message": {
    "subject": "Quarterly Report",
    "from": [{"email": "alice@sender.example", "name": "Alice Smith"}],
    "to": [{"email": "bob@recipient.example"}],
    "size": 15360,
    "bodyValues": {
      "1": {
        "value": "Please find attached the Q4 report...",
        "isEncodingProblem": false,
        "isTruncated": false
      }
    }
  },
  "senderAuth": {
    "spf-mail": "pass",
    "dkim": "pass",
    "dmarc": "pass"
  },
  "client": {
    "ip": "192.0.2.50",
    "ehlo": "mail.sender.example"
  }
}
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "add": [
    {
      "path": "/message/headers",
      "value": {"name": "X-Spam-Status", "value": "No, score=0.5"},
      "index": 0
    }
  ]
}
```

B.1.5. Hook Invocation - Spam Detected

```
POST /v1/hooks/invoke/reg_spam_001 HTTP/1.1
Host: scanner.example.com
Content-Type: application/json
Authorization: Bearer sk_live_abc123
X-MTA-Hooks-Registration: reg_spam_001
X-MTA-Hooks-Request-Id: 01HZ8KAW6D2E3F4G5H6J7K8L9M

{
  "stage": "data",
  "action": "accept",
  "timestamp": "2024-12-21T10:35:00Z",
  "response": {
    "code": 250,
    "enhancedCode": "2.0.0",
    "message": "OK"
  },
  "envelope": {
    "from": {"address": "promo@spammer.invalid", "parameters": {}},
    "to": [{"address": "bob@recipient.example", "parameters": {}}]
  },
  "message": {
    "subject": "YOU HAVE WON $1,000,000!!!",
    "from": [{"email": "winner@lottery.invalid"}],
    "to": [{"email": "bob@recipient.example"}],
    "size": 8192
  },
  "senderAuth": {
    "spf-mail": "fail",
    "dkim": "none",
    "dmarc": "fail"
  },
  "client": {
    "ip": "203.0.113.99",
    "ehlo": "totally-legit.invalid"
  }
}
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "set": [
    {
      "path": "/action",
      "value": "reject"
    },
    {
      "path": "/response",
      "value": {
        "code": 550,
        "enhancedCode": "5.7.1",
        "message": "Message rejected due to spam content"
      }
    }
  ]
}
```

B.2. Outbound Delivery Logging

This example shows outbound hook usage for delivery status logging and compliance.

B.2.1. Registration

```
POST /v1/hooks/register HTTP/1.1
Host: logger.example.net
Content-Type: application/json
Authorization: Bearer sk_live_def456

{
  "name": "Acme MTA mx1",
  "version": "1.0.0",
  "timeoutMs": 5000,
  "serialization": "json",
  "outbound": {
    "stages": ["delivery", "dsn"],
    "properties": null
  }
}
```

```
HTTP/1.1 201 Created
Content-Type: application/json

{
  "registrationId": "reg_logger_001",
  "status": "active",
  "createdAt": "2024-12-21T11:00:00Z",
  "hookEndpoint": "/v1/hooks/invoke/reg_logger_001",
  "negotiated": {
    "serialization": "json",
    "outbound": {
      "stages": ["delivery", "dsn"],
      "properties": ["/envelope", "/message", "/queue", "/server"]
    }
  },
  "endpoints": {
    "deregistration": "/v1/hooks/register/reg_logger_001",
    "status": "/v1/hooks/register/reg_logger_001/status"
  }
}
```

B.2.2. Hook Invocation - Partial Delivery

```
POST /v1/hooks/invoke/reg_logger_001 HTTP/1.1
Host: logger.example.net
Content-Type: application/json
Authorization: Bearer sk_live_def456
X-MTA-Hooks-Registration: reg_logger_001
X-MTA-Hooks-Request-Id: 01HZ8MBN5Q4R3S2T1U0V9W8X7Y

{
  "stage": "delivery",
  "action": "continue",
  "timestamp": "2024-12-21T12:00:00Z",
  "queue": {
    "id": "q_msg_12345",
    "expiresAt": "2024-12-24T12:00:00Z",
    "attempts": 1
  },
  "envelope": {
    "from": {"address": "notify@example.com", "parameters": {}},
    "to": [
      {
        "address": "user1@active.example",
        "parameters": {},
        "status": "delivered",
        "attempt": 1,
        "lastResponse": {
```

```
            "code": 250,
            "enhancedCode": "2.0.0",
            "message": "Delivered"
          }
        },
        {
          "address": "user2@slow.example",
          "parameters": {},
          "status": "deferred",
          "attempt": 1,
          "lastResponse": {
            "code": 451,
            "enhancedCode": "4.7.1",
            "message": "Greylisted, try again"
          },
          "nextAttemptAt": "2024-12-21T12:15:00Z"
        },
        {
          "address": "user3@invalid.example",
          "parameters": {},
          "status": "failed",
          "attempt": 1,
          "lastResponse": {
            "code": 550,
            "enhancedCode": "5.1.1",
            "message": "User unknown"
          }
        }
      ]
    },
    "message": {
      "subject": "System Notification",
      "messageId": "<notify-12345@example.com>",
      "size": 2048
    }
  }

  HTTP/1.1 204 No Content
```

B.3. Deregistration

```
  DELETE /v1/hooks/register/reg_spam_001 HTTP/1.1
  Host: scanner.example.com
  Authorization: Bearer sk_live_abc123
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "registrationId": "reg_spam_001",
  "status": "deregistered",
  "deregisteredAt": "2024-12-21T18:00:00Z"
}
```

Appendix C. Acknowledgments

The authors thank the developers and operators of existing mail filtering systems whose experience informed this design.

Appendix D. Changes

[[This section to be removed by RFC Editor]]

draft-degennaro-mta-hooks-01

* Inverted the registration model: the MTA is now exclusively the HTTP client. The registration, status, and deregistration endpoints reside at the scanner; the MTA no longer exposes any HTTP server to scanners.

* Removed the callback URL concept and callback verification handshake. The scanner’s hook endpoint is returned in the registration response.

* Required X-MTA-Hooks-Registration on every hook invocation. Added X-MTA-Hooks-Request-Id for safe deduplication on retry.

* Specified authentication requirements for hook invocations and credential provisioning practices.

* Specified the error response contract for hook invocations and the status code semantics.

* Clarified action precedence in multi-scanner chains and removed the duplicate Multiple Scanner Handling section.

* Replaced "Modification Order" duplication with a forward reference.

* Documented the failed-silent recipient status.

* Corrected the enhanced status code reference from RFC 3461 to RFC 3463.

* Clarified filter conditions: minSize and maxSize are permitted; the confused reference to Section 4.1.1 of RFC 8621 has been removed.

* Replaced callback-related error codes (CALLBACK_UNREACHABLE, CALLBACK_VERIFICATION_FAILED, TLS_REQUIRED, TLS_CERTIFICATE_INVALID, INVALID_CALLBACK_URL, CALLBACK_NOT_ALLOWED, ALREADY_REGISTERED) with codes appropriate to the new model (REGISTRATION_MISMATCH, REGISTRATION_DEREGISTERED, REQUEST_TOO_LARGE, UNSUPPORTED_PROPERTY, MISSING_REQUEST_ID, SCANNER_UNAVAILABLE).

* Rewrote Security Considerations around the new trust model and credential provisioning.

draft-degennaro-mta-hooks-00

* Initial version

Author’s Address

Mauro De Gennaro
Stalwart Labs LLC
1309 Coffeen Avenue, Suite 1200
Sheridan, WY 82801
United States of America
Email: mauro@stalw.art
URI: https://stalw.art