DCSA Subscription Callback API 1.0

September 2021

Table of contents

Tables

Change history

Rev	Issue	Contributors	Description
1.0	September 2021	DCSA	First release

Terms, acronyms, and abbreviations

Term	Definition
API	Application programming interface
Callback URL	The endpoint that the Subscriber wants the Publisher to send Messages to.
CIA	Confidentiality, Integrity and Authenticity Desired properties in a secure messaging system.
EBL	Electronic Bill of Lading In the context of this document, this is a reference to the standard published by DCSA by the same name.
Event	An Event entity
HMAC	Keyed-hash message authentication code Defined in RFC 2104.
MitM	Man in the Middle. Well defined attacker model for a network attacker.

Term	Definition
Message Bundle	A Message Bundle is one or more Messages submitted to the Subscriber in a single request to the Callback URL.
Message	When an event is registered and there is a subscription matching that event, then Publisher is responsible for informing the Subscriber of the event. This happens by submitting a Message to the Subscriber's callback URL. The actual content of the Message (the request body) is API specific and is beyond the scope of this document. Please refer to the concrete API (such as T&T) for details on this.
OVS	Operational Vessel Schedule In the context of this document, this is a reference to the standard published by DCSA by the same name.
Pending Message	A Message for a subscription that the Publisher has scheduled but the Subscriber has not yet accepted. A pending Message has not yet expired.
Publisher	The one sending Message. In the EBL setting, this is the Carrier.
Shared Secret	A secret shared between the Publisher and the Subscriber. It is used to compute the contents of the **Notification-Signature** header.
Subscriber	The one implementing the Endpoint behind the Callback URL. In the EBL setting, this is the Shipper.

Term	Definition
Subscription	Subscriber can register an interest in certain events by creating a Subscription at the Publisher. The subscription includes a Callback URL, which the endpoint that the Subscriber wants the Publisher to send Messages to.
Subscription ID	This is an opaque ID assigned by the Publisher to a given subscription of type Text(100). This cannot be changed. The security of the protocol does **not** rely on this ID being a secret.
TLS	Transport Layer Security The security protocol used in HTTPS.
T&T	Track and Trace In the context of this document, this is a reference to the standard published by DCSA by the same name.

1 Introduction

This chapter describes the purpose, structure and conventions used in this document.

1.1 Purpose and scope

The DCSA event subscription model aims to make subscription simple to implement for both Publisher and Subscriber. This model is an update to the current DCSA API Design guidelines.

When reviewing this document, please keep in mind that this model is intended for all event subscription systems that DCSA is designing. This implies that the model should satisfy the requirements for EBL, T&T and OVS and the actors in those protocols.

For this reason, we consistently use the generic "Publisher" and "Subscriber" rather than the solution-specific entities (such as "Carrier" and "Shipper" in the EBL context) in this description.

That said, here is a quick overview of the entities for which the protocol was originally intended:

- In EBL via the Document Hub, the Publisher is the Carrier, and the Subscriber is the Shipper
- In T&T, the Publisher is the Carrier, and the Subscriber is the shipper and the consignee.

- In OVS, the Publisher and the Subscriber are carriers and port terminals.

1.2 Pre-requisites

This document assumes the reader is familiar with the DCSA API Design Principles (except the subscription section, which will be superseded by this document). The API Design Principles covers details such as how dates are formatted.

1.3 Conventions

The key words "**MUST**", "**MUST NOT**", "REQUIRED", "SHALL", "SHALL NOT", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**MAY**", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 (https://www.ietf.org/rfc/rfc2119.txt).

2 Goals and non-goals

We define the following goals for this API:

Goal ID	Goal Description
G1	The Subscription Callback API **SHOULD** prefer simplicity for the Subscriber over simplicity for the Publisher.
G2	The Subscription Callback API **MUST** rely on simple validation methods for determining if a Message was submitted by the Publisher.
G3	The Subscriber **MUST** be able to reliably detect and discard fake messages from outsiders without having to contact the Publisher.
G4	Retracted.
G5	The API **MUST** cover how to initiate a "secret rotation" from the Subscriber side.
G6	The API **MUST** describe how to detect freezing attacks, where an MitM deploys a DoS on the Message system. As a side effect, this also enables Subscriber to detect operational issues with their callback endpoints in a timely manner.

Goal ID	Goal Description
G7	The API SHOULD ensure that a third-party cannot cause Subscribers to perform a DoS/DDoS attack against the Publisher APIs.

Table 1: The goals of the API

There are also some non-goals:

- The API spec only covers the Integrity and Authenticity of the CIA security requirements. For Confidentiality, the spec relies on the callback using TLS (https).

- The API makes no guarantee of security while any of the following statements are true:
  - The Shared Secret is leaked to a third party (either by accident or if either side is hacked). In this case, the third party can send fake Messages to the Subscriber that the Subscriber will believe came from the Publisher.
  - A third-party gains access the Publisher's subscription system and alters the Subscription of the Subscriber. In this case, the third party can prevent the Subscriber from receiving Messages. The third-party is also able to insert itself as a Man-in-the-Middle by replacing the callback URL of the Subscription. They will not be able to inject fake Messages, but they can read the Messages, deliberately delay, or withhold any Messages of their choosing.

  Note that the API does provide the means for rotating the Shared Secret after it has been leaked. This enables parties to easily recover from a leak provided the callback URL is still valid. In some cases, the Subscriber may want to change the callback URL while changing the secret as well.

- The authentication and authorization for CRUD operations in the Publisher's Subscription API provided by the Publisher is beyond the scope of this document.
  - Note that given the previous bullet, this is a necessity for security of the entire system. The Publisher MUST choose an authentication and authorization model with this in mind, but this document does not mandate which models to use for this purpose.

- The API spec does not mandate security for storing the Shared Secret. But to implement this protocol both the Publisher and Subscriber will have to save it in a reversible way as it is necessary to submit and verify the Message.

3 High level overview of the callback API

The subscription setup follows the steps:

1) The Subscriber sets up a subscription in the Publisher's system (POST …/event-subscriptions)
   - The subscription **MUST** include a callback URL, from where the Subscriber accepts the callback.
   - The subscription **MUST** include a Shared Secret. as a part of the body (see **3.3 Exchanging secrets between Publisher and Subscriber**).
2) The Publisher confirms the subscription and returns the Subscription ID to the Subscriber.
3) The Subscriber records the Subscription ID associated with their Shared Secret.

When sending the message or validating the endpoint:

- The Publisher **MUST** perform a POST (when sending a Message) or HEAD (when validating the endpoint is available) to the callback URL provided by the Subscriber exactly as it is provided by the subscriber.
- The Publisher **MUST** include the Subscription-ID header containing the Subscription ID on POST calls.
  - Please see **3.1.1 Verification of callback URL** for handling the validation of the callback URL while creating a subscription.
- The Publisher **MUST** sign the Message Bundle and put the signature in the **Notification-Signature** header. The description of how this is computed is covered in **3.2 The signature provided in the Notification-Signature header**.
- When POST'ing a Message Publisher **MUST** include the Message as the HTTP body in JSON format with the Content-Type header set to "application/json".

3.1 Validity check of Message on Subscriber side

On receiving a POST request, the Subscriber **MUST** perform the following checks (it **SHOULD** do it in the listed order):

1) The request **MUST** contain the HTTP header **Notification-Signature** header, which contains the signature (validated in a later step).

2) The request header **MUST** include the Subscription-ID header and it **MUST** be a String.

3) The Subscriber is **RECOMMENDED** to perform additional validation of the request such as validation of custom embedded values in the callback URL that Subscriber use in their subscriptions if any.

4) The Subscriber fetches the Shared Secret for the given Subscription-ID and computes the signature of the request payload (described in **3.2 The signature provided in the Notification-Signature header**). (G1, G2, G3)

5) The **eventCreatedDateTime** **MUST** be in the past.

    - Subscriber **MAY** accept **eventCreatedDateTime** values that are a few seconds into the future to account for minor time synchronization issues.

If any of these checks fail, the message **MUST** be rejected as invalid unless otherwise stated in the check. (G7) Validation of the Subscription ID and **Notification-Signature** **SHOULD** cause an HTTP 401 Unauthorized response. For other validation checks that cause a failure, HTTP 400 Bad Request is **RECOMMENDED** as default when no other HTTP codes are more applicable.

It is **RECOMMENDED** that the Subscriber defer parsing the message body until after these checks, as it minimizes the attack surface (e.g., in case of security bugs in the underlying JSON parser).

3.1.1 Verification of callback URL

Upon subscription creation, the publisher **MUST** validate the callback endpoint; this can be done by calling the endpoint with a HEAD request to verify the existence of the endpoint. If the endpoint fails, the creation of the subscription **MUST** fail. If callback endpoint responds with a 204 HTTP status code, the check succeeds, and the endpoint is considered available.

The Publisher **MUST** omit the **Notification-Signature** header (HEAD requests have no body to be signed) and the **Subscription-ID** header. The Publisher **SHOULD** ensure that these headers are absent.

The Publisher **MAY** impose additional vendor-specific requirements for callback URLs – examples include ensuring the hostname or domain name in the callback URL is on an approval list managed by the Publisher.

The Subscriber is **RECOMMENDED** to perform additional validation of the HEAD request such as validation of custom embedded values that Subscriber uses in their subscriptions when evaluating a verification callback. This ensures that a subscription fails if the custom embedded values are invalid.

3.1.2 Recommendations for additional security on the callback URL

It is in the Subscriber's interest to reject false messages as early as possible. This section covers some approaches the Subscriber can and is **RECOMMENDED** to use if they make sense for the given Subscriber.

- Use unguessable Callback URLs such as https://callback.example.com/callback/$RANDOM_STRING. This approach requires the Subscriber to keep track of which values are valid for $RANDOM_STRING, but in return it can be used as a quick rejection filter. For this to work, the callback URL will have to be kept secret and the Subscriber is **RECOMMENDED** to change the callback URL when the secret is changed.
- Use IP or reverse DNS based approval listing to restrict which IPs or hostnames that can access the Subscriber's callback endpoint. This solution is only applicable if the Publisher provides stable IP ranges or hostnames for the servers publishing the messages.
- Enforce an upper limit on the payload size. This limit **SHOULD** be aligned with the Publisher and their expected max payload size as to avoid incorrectly rejecting valid messages from the Publisher.
- Use IP-based request rate limiting to reduce the number of parallel requests from a given source. This is most useful when combined with the payload size limit.

3.2 The signature provided in the Notification-Signature header

Every Message is signed via the **Notification-Signature** header. The header has the following format:

```
Notification-Signature: <signature-type>=<signature>
```

The **<signature-type>** part is a keyword that determines which algorithm was used to compute the signature. Currently only one signature type is defined (**sha256**). The **<signature>** part is the signature itself encoded in hexadecimals.

The signature **MUST** cover the entire request body of the request including whitespace and newlines. The content **MUST** be decoded into bytes using the UTF-8 encoding before computing the signature. *None* of the HTTP headers nor the request URL is covered by the signature.

3.2.1 The sha256 signature type

The sha256 signature type is computed as an HMAC-SHA256 over the Message (the request body). The Subscriber **MUST** provide a Shared Secret of at least 32 bytes that is used as the key for the HMAC computation. The key size SHOULD NOT be larger than 64 bytes as keys beyond 64 bytes do not provide additional security for the HMAC-SHA256 algorithm.

The algorithm is like the one used by GitHub for their webhooks, and their examples can *almost* be used as-is. There are a few minor differences, such as the Header name and requirements to the secret length. The GitHub webhook security document can be found at https://docs.github.com/en/developers/webhooks-and-events/webhooks/securing-your-webhooks

3.2.2 A concrete example of the signature using the sha256 method.

This section is informative with the aim of clarifying how the signature is computed.

The following JSON Message is a concrete example:

```
{
  "eventID": "84db923d-2a19-4eb0-beb5-446c1ec57d34",
  "eventType": "SHIPMENT",
  "eventDateTime": "2019-11-12T07:41:00+08:30",
  "eventClassifierCode": "ACT",
  "eventTypeCode": "ARRI",
  "shipmentID": "123e4567-e89b-12d3-a456-426614174000",
  "shipmentInformationTypeCode": "SRM"
}
```

The same document encoded in base64 for those who want to reproduce the signature (to avoid ambiguity about caused by formatting):

```
ew0KICAiZXZlbnRJRCI6ICI4NGRiOTIzZC0yYTE5LTRlYjAtYmViNS00NDZjMWVjNTdkMzQiLA0K
ICAiZXZlbnRUeXBlIjogIlNISVBNRU5UIiwNCiAgImV2ZW50RGF0ZVRpbWUiOiAiMjAxOS0xMS0x
MlQwNzo0MTowMCswODozMCIsDQogICJldmVudENsYXNzaWZpZXJDb2RlIjogIkFDVCIsDQogICJl
```

```
dmVudFR5cGVDb2RlIjogIkFSUkkiLA0KICAic2hpcG1lbnRJRCI6ICIxMjNlNDU2Ny1lODliLTEy
ZDMtYTQ1Ni00MjY2MTQxNzQwMDAiLA0KICAic2hpcG1lbnRJbmZvcm1hdGlvblR5cGVDb2RlIjog
IlNSTSINCn0=
```

And the 32-byte key (Shared Secret):

```
1234567890abcdef1234567890abcdef
```

Then the signature is (in hexadecimals):

```
8909e231195705fec82bfa55e839cb76a8ceffe24a13e79256801179b9a9c7a0
```

The Notification-Header would be (with the middle part of the signature truncated for visual reasons):

```
Notification-Signature: sha256=ae688919f5e31f4c210ca6a…21a8507ee395de5e2de
```

The following snippet of Java code can be useful for reproducing the signature:

```
public static byte[] computeSignature(byte[] secretKey, byte[] payload) throws
Exception {
    final String javaAlgorithmName = "HmacSHA256";
    Mac mac = Mac.getInstance(javaAlgorithmName);
    mac.init(new SecretKeySpec(secretKey, javaAlgorithmName));
    return mac.doFinal(payload);
}

public static void sampleSignature() throws Exception {
    byte[] key =
"1234567890abcdef1234567890abcdef".getBytes(StandardCharsets.UTF_8);
    byte[] payload = Base64.getDecoder().decode(

"ew0KICAiZXZlbnRJRCI6ICI4NGRiOTIzZC0yYTE5LTRlYjAtYmViNS00NDZjMWVjNTdkMzQiLA0K" +

"ICAiZXZlbnRUeXBlIjogIlNISVBNRU5UIiwNCiAgImV2ZW50RGF0ZVRpbWUiOiAiMjAxOS0xMS0x" +

"MlQwNzo0MTowMCswODozMCIsDQogICJldmVudENsYXNzaWZpZXJDb2RlIjogIkFDVCIsDQogICJl" +

"dmVudFR5cGVDb2RlIjogIkFSUkkiLA0KICAic2hpcG1lbnRJRCI6ICIxMjNlNDU2Ny1lODliLTEy" +

"ZDMtYTQ1Ni00MjY2MTQxNzQwMDAiLA0KICAic2hpcG1lbnRJbmZvcm1hdGlvblR5cGVDb2RlIjog" +
                    "IlNSTSINCn0="
    );
    byte[] signature = computeSignature(key, payload);

    System.out.println("Notification-Signature: sha256=" +
Hex.encodeHexString(signature));
    System.out.println(" --- 8< --- PAY LOAD (" + payload.length + " bytes) ---
```

```
8< ---");
    System.out.println(new String(payload));
    System.out.println(" --- 8< --- PAY LOAD --- 8< ---");
}
```

3.3 Exchanging secrets between Publisher and Subscriber

All secrets are provided by the Subscriber and submitted to the Publisher via the Publisher's subscription API. As the secrets need to be byte string, they **MUST** be submitted in base64 encoding inside the request. The Subscription API **MUST** accept Shared Secret via the following endpoints:

- In the **POST …/event-subscriptions** to create a new subscription via the secret field.
- In the **PUT …/event-subscriptions/{subscriptionID}/secret** endpoint via the secret field. This is used for resetting the Shared Secret on an existing subscription.

The Publisher **MUST NOT** expose the shared secret in any of the API endpoints – it is entirely "write-only".

The Publisher **MUST** reject requests containing secrets that are not adequate for the signature algorithm (e.g. too short or too long). The Publisher **MUST** perform the following checks on receiving the secret:

1. Ensure that the received secret can be decoded as a base64 string.
2. Ensure that the decoded secret matches length requirements for the secret (see Error! Reference source not found.)

3.4 Subscription Management API

The Publisher **MUST** provide the following event subscription endpoints:

- **POST …/event-subscriptions** to create a new subscription.
- **GET …/event-subscriptions** to list all subscriptions the caller has access too.
- **GET …/event-subscriptions/{subscriptionID}** to provide details about a concrete subscription.
- **PUT …/event-subscriptions/{subscriptionID}** to update a particular subscription.
- **PUT …/event-subscriptions/{subscriptionID}/secret** to update the secret for a particular subscription.
- **DELETE …/event-subscriptions/{subscriptionID}** to remove (cancel) a concrete subscription.

The following attributes are reserved for the event subscription API:

Attribute	Description	Example
subscriptionID	The subscription ID. The ID is generated by the Publisher when the subscription is created.	784871e7-c9cd-4f59-8d88-2e033fa799a1
secret	The shared secret, which is a base64 encoded byte string blob during transfer. See **3.3Exchanging secrets between Publisher and Subscriber** for which endpoints it is applicable to.	MTIzNDU2Nzg5MGFiY2RlZjEyMzQ1Njc4OWFiY2RlZg==

The subscription **MAY** have additional attributes specific to the concrete event types managed to enable filtering on said attributes - such as a "carrierBookingReference" attribute to denote that the Subscriber only wants to receive events related to that concrete booking reference. However, these are specific to the concrete solution being implemented and therefore beyond the scope of this document.

4 Non-delivery of Messages

When the Publisher attempts to deliver a Message to the Subscriber, there are three overall scenarios:

- The Publisher successfully connects to the Subscriber endpoint and the Subscriber responds with a successful HTTP response (code 204) indicating that the Message was accepted. The Publisher records the Message as delivered and removes it from its internal queue of Pending Messages for the subscription.
- The Publisher successfully connects to the Subscriber endpoint and the Subscriber responds with a non-successful response (any code other than 204). In this case, the Publisher **MUST** attempt to reschedule the Message to be retried later after the delay defined by 4.1 Retry .
- The Publisher is unable to connect or deliver the Message to the Subscriber's endpoint. This covers all causes not covered by the above (including connection timeouts, a TCP reset error, etc.). The Publisher **MUST** handle this as if they had received a HTTP 503 "Service Temporarily Unavailable" response *without* a **Retry-After** header.

4.1 Retry policy

If the request was not a success (HTTP 204), the Publisher **MUST** retry sending the event. If a **Retry-After** header is present in the response – this **SHOULD** be honored. The Subscriber **SHOULD** use the **Retry-After** header in their response in 413, 429 and 503 responses where they can reliably estimate when they will accept the next Message delivery attempt. Otherwise, it is **RECOMMENDED** that an exponential back-off retry policy is used. An example of this could be to retry after 1 min, 2 min, 4 min, 8 min, etc.

It is **RECOMMENDED** to have an upper bound on the retry deadline – either in the form of letting Messages expire (see **4.2 Expiry**) or simply an upper limit on the retry frequency. Once the retry frequency passes once per day or once per week, higher delays are unlikely to make a difference.

Every time a publisher needs to retry sending an event, the publisher **MUST** make sure the Shared Secret used is updated from the subscription. Cached Shared Secrets **MUST NOT** be used since the Shared Secret **MAY** have been updated since last retry.

The Publisher **MAY** reduce or clamp the retry delay when a new Message is submitted, except when the delay originated from a **Retry-After** header. This is a tradeoff between attempting timely delivery of new Messages vs. using resources to attempt to connect to an endpoint that is unavailable or overloaded.

The Publisher **SHOULD** keep HTTP 413 (Payload Too Large) and **3.1.2** in mind when bundling multiple Messages into one request.

4.2 Expiry

The Publisher **MAY** let Pending Messages expire after a period. When a Pending Messages expires, the Publisher discards the notification from their queue of Pending Messages. The exact deadline is implementation defined but **SHOULD** reflect how relevant the message is at that point.

The Publisher **MUST NOT** expire a Pending Messages before the deadline even if the retry policy implies it will not be retried before the expiry occurs. The rationale for this is that the delay can be reset due to other events or actions.

4.3 Persistent network issues and freeze attacks

Network communication is unreliable by nature, some causes may be:

- Subscriber's callback endpoint is unreachable for the Publisher due to undetected operational issues on either side – expired TLS certificates, mistake in firewall rules, logic bug in the implementation, etc.
- An MitM attacker that deliberately denies the request.

No specification can prevent any of these issues from occurring. However, in a callback scenario covered by this specification, there is an additional issue. Namely, that it is impossible to tell if an absence of Messages is caused by an issue, or because there were no Messages to send. Particularly because the Subscriber cannot make assumptions about connectivity to the Publisher's Subscription API, as those servers are not necessarily the ones sending Messages.

To solve this, the Subscriber SHOULD schedule a periodic poll for updates from the Publisher via the Publisher's Event API (that is associated with the subscription API). This specification does not mandate a polling frequency – the Subscriber is expected to choose a reasonable frequency that both matches the expected frequency of the related events and does not put unreasonable load on the Publisher's infrastructure. (G6)

If the Subscriber concludes that there has been a persistent connectivity issue, the Subscriber is RECOMMENDED to update the Shared Secret for the subscription, as this will prompt the Publisher to attempt to resend any Pending Messages.

5 Rotation of the Shared Secret

The following describes the protocol for revoking / rotating the Shared Secret in any case, regardless of reason. (G5)

It is a healthy practice to change secrets regularly. The Subscriber is RECOMMENDED to implement a scheduled or periodic rotation of the Shared Secret. If the Subscriber wishes to initiate a rotation of the Shared Secret, then the Subscriber performs an update of the Subscription via the Publisher's Subscription API.

When the Publisher confirms an updated Shared Secret upon subscription, then:

1) If the Publisher supports Publisher initiated Shared Secret rotations, then the Publisher MUST mark the subscription as no longer needing a Shared Secret rotation.

2) The Publisher SHOULD reset the retry delay for any Pending Messages beyond an implementation defined threshold. It is RECOMMENDED that delays beyond an hour are reset to at most an hour. (G6)