E. Rescorla
Windy Hill Systems, LLC
B. Lassey
Google
Internet-Draft
Intended status: Informational
Expires: 25 March 2024
22 September 2023

Transferring Digital Credentials with HTTP
draft-rescorla-tigress-http-00

Abstract

There are many systems in which people use "digital credentials" to control real-world systems, such as digital car keys, digital hotel room keys, etc. In these settings, it is common for one person to want to transfer their credentials to another, e.g., to share your hotel key. It is desirable to be able to initiate this transfer with a single message (e.g., SMS) which kicks off the transfer on the receiver side. However, in many cases the credential transfer itself cannot be completed over these channels, e.g., because it is too large or because it requires multiple round trips. However, the endpoints cannot speak directly to each other and may not even be online at the same time. This draft defines a mechanism for providing an appropriate asynchronous channel using HTTP as a dropbox.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://ekr.github.io/draft-rescorla-tigress-http/draft-rescorla-tigress-http.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-rescorla-tigress-http/.

Discussion of this document takes place on the Transfer dIGital cREdentialS Securely Working Group mailing list (mailto:tigress@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/tigress/. Subscribe at https://www.ietf.org/mailman/listinfo/tigress/.

Source for this draft and an issue tracker can be found at https://github.com/ekr/draft-rescorla-tigress-http.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

DISCLAIMER: This draft is work-in-progress and has not yet seen significant (or really any) security analysis. It should not be used as a basis for building production systems.

There are many systems in which people use "digital credentials" to control real-world systems, such as digital car keys, digital hotel room keys, etc. Generally these are proprietary system-specific credentials are embedded in and used by a (potentially proprietary) mobile app. In these settings, it is common for one person to want to transfer their credentials to another, e.g., to share your hotel key with a family member.

Although the credentials and transfer mechanisms are often proprietary they share a common workflow in which:

1. The Sender initiates the transfer from their app and sends an invitation message over a preexisting channel such as SMS or e-mail.

2. Bob receives the invitation message from Alice and hands it to his app (ideally this would happen automatically, e.g., by some URL handler).

3. Bob uses the invitation message to contact Alice to complete the transfer. This may require multiple round trips between Alice and Bob. In addition, Alice or Bob may need to contact some external server, but this is out of scope for this protocol.

The preexisting channel may not be suitable for completing the transfer, for instance because it has insufficient bandwidth. or because it requires manual intervention by the users. In addition, the participants may not be online simultaneously, so a "store-and-forward" channel is required. [I-D.ietf-tigress-requirements] describes the requirements in more detail. This document specifies how to build such a channel using a standard HTTP [RFC9110] server.

2. Conventions and Definitions

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.

3. Overview of Operation

Figure 1 provides a broad overview of the message flow:

```
Alice                      HTTP Server                   Bob

Initiating (R) --------------------------------------------->
PUT <L0> MSG0 ---------------->
                               <-------------------- GET <L0>
                               <----------------- DELETE <L0>
                               <--------------- PUT <L1> MSG1
GET <L1> --------------------->
<------------------------- MSG1
DELETE <MSG1> ---------------->
                              ...
```

Figure 1: Overview of Operation

In order to initiate the transfer, Alice generates a random secret value R. She then does the following:

1. Sends R and the address of the HTTP server to Bob over the preexisting channel.

2. Generates the first protocol message MSG0 and stores it in a location on the HTTP server (L0) pseudorandomly generated from R.

When Bob receives the initiating message, he uses R to determine L0, retrieves it from the server, and then deletes it. In order to send a message (MSG1) to Alice, Bob stores it at a new pseudorandom location L1 (again, based on R). Alice retrieves it and then deletes it. Any further message exchanges proceed in the same fashion.

4. Architectural Model

The overall system has the following architecture:

```
+------------------------------------------------+
|   Credential Exchange Protocol (proprietary)   |
+------------------------------------------------+
|    Protected Message Format (Section TODO)     |
+------------------------------------------------+
|           HTTP Binding (Section TODO)          |
+------------------------------------------------+
```

The lowest level of operation is a binding to HTTP specifying how to use an HTTP server as a store-and-forward channel, specified in Section 6. That channel is then used to carry encrypted messages in the format defined in Section 7. Those messages contain an opaque payload that is used by the relevant proprietary credential exchange protocol.

5. Initiating Message

The initiating message needs to contain at least the following three values:

* A URI template. This MUST contain a single variable, named "tigress_location". [[TODO: Need to flesh this out some more.]] This template MUST be for an HTTPS URI.

* A secret value R generated with a cryptographically secure PRNG [RFC4086] and containing at least 256 bits of entropy.

* The AEAD algorithm defined using TLS 1.3 cipher suites Section 8.4 of [RFC8446].

In practice, it will probably contain other information such as the type of credential to be transferred and perhaps some human-readable context. These values are out of topic for this specification.

The initiating message SHOULD be delivered over a secure channel but this protocol provides limited security even when that does not happen (see Section 8).

6. HTTP Binding

The basic concept of the HTTP binding is very simple. In order for endpoint A to send a message to endpoint B, A does a PUT to a resource in a predefined secret location. B then does a GET to retrieve the resource and a DELETE to remove it. Receivers MUST delete messages immediately after they have retrieved them.

[[OPEN ISSUE: Polling is bad, so we're going to need some kind of notification mechanism, but this document doesn't specify that.]]

HTTP requests MUST not contain information from other context (e.g., browser cookies). [[OPEN ISSUE: Can it contain other authentication information, for instance for attestation.]]

The URL for message i is generated as follows, using the HKDF-Expand-Label function from TLS 1.3 [RFC8446].

```
U_i = HKDF-Expand-Label(R, "Location",
                        Transcript, 256)
```

[[OPEN ISSUE: This construction puts some secret information (the nonces from the previous messages) in the transcript. Maybe we should instead do a combiner?]]

Where "Transcript" is the concatenation of the plaintext of all previous messages and HKDF-Expand-Label uses the hash from the defined cipher suite.

The URL is then generated by subsituting the URL-safe base64 encoding [RFC4648] for the "tigress_location" variable in the URL template.

[[OPEN ISSUE: What is the media type of the message?]]

HTTP servers used for this protocol MUST NOT allow enumeration of resources that match the URL template.

This protocol operates in a lock-step "ping-pong" fashion. Each endpoint can send exactly one message and then must wait for the other side to reply before sending another. The sender of the credential speaks first.

7. Message Format

All messages are encrypted using the AEAD algorithm specified by the cipher suite, formatted as an O-HTTP "Encapsulated Response" Section 4.2 of [I-D.ietf-ohai-ohttp]). The "nonce" MUST be pseudorandomly generated.

The encryption key is generated as follows:

```
    K_i = HKDF-Expand-Label(R, "Key",
                            Transcript, 256)
```

The plaintext of the message is as follows (using TLS syntax):

```
struct {
  opaque random<0..255>;
  uint16 message_id;
  opaque message<0..2^32-1>;
} TigressPlaintext;
```

These fields have the following values:

random A cryptographically random field. The first message in each direction MUST have a random value of at least 16 octets. Subsequent messages MAY contain random values of at any length.

message_id The sequence number of the message, starting from 0 and incrementing with each message in the exchange. This space is shared and so in practice even numbers are from the credential sender and odd numbers from the receiver. [[OPEN ISSUE: Do we need this? It’s basically a double check because the system guarantees uniqueness.]]

message The proprietary credential exchange message.

Upon receiving a message, an endpoint MUST first deprotect it using the correct key and algorithm. If AEAD deprotection fails, it MUST signal an error and abort the protocol run.

Endpoints MUST check that the message_id has the expected value and that the random values are of the right length must signal an error and abort the protocol run if they are incorrect.

8. Security Considerations

The protocol is intended to guarantee the following properties:

1. In order to determine the location of a message, an entity must know both R and the plaintext of every previous message.

2. In order to decrypt a message, an entity must know both R and the plaintext of every previous message.

If R is delivered over a secure channel, then an attacker should not be able to read any message or inject a new one. Because the HTTP server sees messages when they are stored it can delete them or replace them with an invalid message, but because it does not have R it cannot generate a new valid message or replay an old one. The result of this attack is to cause the credential exchange to fail. An attacker other than the server does not know the location of the resource and therefore cannot even store bogus values. If the

An attacker who learns R prior to the protocol exchange can simply impersonate the receiver. This is why R should be sent over a secure channel. If it is necessary to send R over an insecure channel then some other mechanism is required to prevent this attack. [[OPEN ISSUE: this is not great, but it seems to be the assumed setting based on list discussion.]]

An attacker who learns R after the receiver has retrieved and and deleted the first message will not have the random value from MSG0 and therefore will not be able to determine either the location and encryption key for MSG1, so cannot forge their own message to the sender or any future message. Note that an attacker who learns R after the receiver has retrieved MSG0 but before they have deleted it and replied can race the receiver to respond. If they win the race, then they will be able to complete the protocol exchange with the sender and the receiver will be locked out. This is why it is important for the receiver to delete MSG0 immediately upon retrieval.

The reason for including the transcript of all previous messages in the next key and URL is that it straightforwardly includes the random values which each side must send in their first message. It also serves to bind each message to those that came before it, though this does not have a straightforward security rationale. Note that if any message is lost, then the entire exchange fails and so the HTTP server is assumed to be reliable. This is one reason why the delete is explicit rather than a side effect, thus avoiding issues where the retrieval of a message fails but the server thinks it succeeded and deletes the message.

9. IANA Considerations

This document has no IANA actions.

10. References

10.1. Normative References

[I-D.ietf-ohai-ohttp]
Thomson, M. and C. A. Wood, "Oblivious HTTP", Work in Progress, Internet-Draft, draft-ietf-ohai-ohttp-10, 25 August 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-ohai-ohttp-10>.

[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>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/rfc/rfc4086>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/rfc/rfc4648>.

[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>.

[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>.

[RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/rfc/rfc9110>.

10.2. Informative References

[I-D.ietf-tigress-requirements] Vinokurov, D., Astiz, C., Pelletier, A., Karandikar, Y., and B. Lassey, "Transfer Digital Credentials Securely - Requirements", Work in Progress, Internet-Draft, draft-ietf-tigress-requirements-00, 9 August 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-tigress-requirements-00>.

Acknowledgments

Thanks to Chris Wood and Martin Thomson for helpful discussions.

Authors' Addresses

Eric Rescorla
Windy Hill Systems, LLC
Email: ekr@rtfm.com

Brad Lassey
Google
Email: lassey@google.com

D. Vinokurov
Y. Karandikar
M. Lerch
A. Pelletier
Apple Inc
N. Sha
Alphabet Inc

11 January 2024

Transfer Digital Credentials Securely
draft-vinokurov-tigress-http-00

Abstract

Digital Credentials allow users to access Homes, Cars or Hotels using their mobile devices. Once a user has a Credential on a device, sharing it to others is a natural use case. Process of sharing credentials should be secure, privacy preserving and have a seamless user experience. To facilitate Credential sharing, a new transport is required. This document defines that new transport to meet unique requirements of sharing a Credential.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://datatracker.ietf.org/doc/draft-vinokurov-tigress-http/. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-vinokurov-tigress-http/.

Discussion of this document takes place on the Transfer dIGital cREdentialS Securely Working Group mailing list (mailto:tigress@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/tigress/. Subscribe at https://www.ietf.org/mailman/listinfo/tigress/.

Source for this draft and an issue tracker can be found at https://github.com/dimmyvi/tigress.

Status of This Memo

This Internet-Draft will expire on 14 July 2024.

Copyright Notice

Table of Contents

1. Introduction

Mobile devices with ever increasing computational power and security capabilities are enabling various use cases. One such category includes use of mobile devices to gain access to a property that a user owns or rents or is granted access to. The cryptographic material and other data required to enable this use case is termed as Digital Credential.

Based on type of property, various public or proprietary standards govern details of Digital Credentials. These sets of standards and the bodies setting them are collectively termed as Verticals. The details include policies, mechanism and practices to create, maintain and use Digital Credentials. The process of getting a Digital Credential on a mobile device is termed as Provisioning.

Once a user has a Digital Credential provisioned on their mobile device, sharing it to others is a natural use case. Sharing a Credential should feel like a natural extension of regular communication methods (like instant messaging, sms, email). The user experience of sharing a Credential should be intuitive, similar to sharing other digital assets like photos or documents. The sharing process should be secure and privacy preserving.

Credentials pose two unique requirements that differ from sharing other digital assets. The Initiator and Recipient devices may need to communicate back and forth to transfer the necessary Provisioning Information. The Provisioning Information exchange must be limited to Initiator device and the first Recipient device to claim the information.

To achieve these goals, a new transport is necessary. This document defines HTTP [RFC9110] based API to create such a transport, termed as Relay Server. The document also defines data in JSON standard [RFC8259] to enable a uniform user experience for securely sharing Digital Credentials of various types.

2. Conventions & Definitions

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.

2.1. General Terms

* Digital Credential (or simply Credential) - Cryptographic material and other data used to authorize User with an access point. The cryptographic material can also be used for mutual authentication between user device and access point.

* Digital Credential Vertical (or simply Vertical) - The public or proprietary standards that that define details of Digital Credentials for type of property accessed. The details include policy, process and mechanism to create, maintain and use Digital Credentials in the given Vertical.

* Provisioning - A Vertical defined process of adding a new Digital Credential to the device.

* Provisioning Entity - An entity that facilitates creation, update and termination (Lifecycle Management) of the Credential. Based on Vertical, the role of Provisioning Entity may be played by various actors in various stages of Credential lifecycle.

* Provisioning Information - data transferred from Initiator to Recipient that is both necessary and sufficient for the Recipient to Provision a Credential.

* Initiator - User and their device initiating a transfer of Provisioning Information to a Recipient.

* Recipient - User and their device that receives Provisioning Information and uses it to provision a new Credential.

* Relay Server - an intermediary server that provides a standardized and platform-independent way of transferring Provisioning Information between Initiator and Recipient, acting as a temporary store and forward service. This is the new transport defined by this document.

* Secret - a symmetric encryption key shared between an Initiator and Recipient device. It is used to encrypt Provisioning Information stored on the Relay server.

3. Sharing Process

3.1. Some Example Use Cases

* Amit owns a car that supports Digital Credentials. Being a tech enthusiast, he has the Credential provisioned on his mobile device. Amit can now use his mobile device to lock/unlock and operate his car. One Monday he is out of town and realizes that his car needs to be moved for street cleaning. He asks his neighbor Bob for help via their favorite instant messaging method. As Bob agrees, Amit shares the Digital Credential to Bob via the next instant message. Bob accepts the Credential and uses his mobile device to unlock Amit's car and drive it to the other side of street.

* Alice booked a room at a hotel that supports Digital Credentials. Being a frequent traveller, she has the Digital Credential provisioned on her mobile device. As her flight gets delayed, she realizes that her partner Bakari will reach the hotel first. So she shares the Digital Credential with him over email. Bakari sees the email after his flight lands and he accepts the shared Credential. On his arrival to the hotel, Bakari is able to access common areas and their room using his mobile device.

3.2. Credential Sharing Flow

A simplified sharing flow is shown in the sequence diagram below. Initiator (User) sends an invitation to share a Credential over their preferred communication method. Recipient (user) accepts the Credential share invitation. Then the two devices go back and forth as necessary to transfer Provisioning Information without further interaction by user. After the transfer is complete Recipient device gets the Credential Provisioned according to Vertical defined process.

"

"

Initiator Device Relay Se
rver Recipient Device
Initiator User
Recipient User
Initiate Credential Share

>

create a mailbox, establish Initiator clai
m

>

Invitation to accep
t Credential
over IM, sms, emai
l etc
>

accept the Credential
<

establish Recipient claim
<

LOOP Transfer

request Provisioning Information
<

Forward request
<

Provisioning Information response
>

forward response
>

Finish Credential Provisioning
Initiator User Recipien
t User
Initiator Device Relay Se

```
rver                  Recipient Device
            "
            "
```

3.3. Relay Server Design Requirements

Based on the sharing flow, we can see that Relay server is an important component of the credential Sharing flow. Relay server design needs to adhere to following requirements :

* Relay server SHALL provide confidentiality and integrity to the transfer of Provisioning Information.

* Transfer of Provisioning Information MAY require several round trips. Relay server SHALL guarantee round trip communication between initiator device and first device to establish Recipient claim.

* Relay Server SHALL support flow of information that MAY NOT always be in turn taking fashion. Same party SHALL be allowed to send back to back messages. E.g. a cancel message may be sent by same party that sent the previous message.

* User involvement in the process needs to be minimal for a seamless user experience. A lay user is expected to be unaware of Relay Server (similar to any transport protocols like TCP/IP). So Relay Server SHALL be able to function without user interaction.

* Initiator and Recipient MAY NOT be online at the same time. So Relay Server SHALL be able to store and forward data. It is RECOMMENDED to have notification mechanism for snappy user experience.

* To protect user privacy, Relay server SHALL NOT require any identifying information of the 2 parties involved in the transfer.

* Relay Server SHALL allow encrypted data (that can not be deciphered by the Relay Server itself) to be stored and transferred.

* Relay Server MAY host multiple mailboxes at the same time, each bound to various pairs of Initiator and Recipient Devices. Relay Server SHALL not be able to relate the devices across various mailboxes.

* User preferred communication methods need to be allowed for invitation delivery. It’s assumed that user is familiar with them and trusts them to be secure enough to deliver messages to intended recipient. But security properties of the methods can not be taken for granted in the design of the Relay Server. Verticals can require second factor to authenticate Recipient if they deem it necessary. Policies and mechanisms for this second factor are in the realm of the Verticals and outside the scope of this document.

4. Relay Server Design

4.1. Design Elements

* Mailbox - A place to store data on the Relay Server. A reader can also read the data from mailbox.

* MailboxIdentifier - a unique identifier for the given mailbox, generated by the Relay server at the time of mailbox creation. The value is a UUID [RFC4122].

* Device Claim - a unique token allowing the caller to read from / write data to the mailbox. Initiator Device generates a Device Claim and presents it to the Relay Server at time of mailbox creation. Relay server creates a mailbox, and binds it to Initiator's Device Claim. Recipient Device generates a Device Claim and presents it to the Relay Server, at time of first read from mailbox. Relay server binds the mailbox to the Recipient's Device Claim. Thus, both Initiator and Recipient devices are bound to the mailbox (allowed to read from / write to it). Only Initiator and Recipient devices that present valid Device Claims are allowed to send subsequent read/update/delete calls to the mailbox. The value SHALL be a unique UUID [RFC4122]. Initiator and Recipient MUST use different values for Device Claim. Implementation SHOULD assign unique values for new mailboxes (avoid re-using values).

* Notification Token - a short or long-lived unique token stored by the Initiator or Recipient Device in a mailbox on the Relay server. This allows Relay server to send a push notification to the Initiator or Recipient Device, informing them of updates in the mailbox.

4.2. API connection details

The Relay server API endpoint MUST be accessed over HTTP using an https URI [RFC2818] and SHOULD use the default https port. This ensures confidentiality property of the transfer process.

Request and response bodies SHALL be formatted as either JSON or HTML (based on the API endpoint). The communication protocol used for all interfaces SHALL be HTTPs.

All Strings SHOULD be UTF-8 encoded (Unicode Normalization Form C (NFC)).

An API version SHOULD be included in the URI for all interfaces. The version at the time of this document's latest update is v1. The version SHALL be incremented by 1 for major API changes or backward incompatible iterations on existing APIs.

4.3. Sharing Flow With API calls

Sequence diagram below shows an example sharing flow with detailed API calls.

* Initiator Device composes Provisioning Information and encrypts it with a Secret before storing it in a mailbox on Relay Server

* Initiator Device generates a unique token - an Initiator Device Claim - to be sent to Relay Server. Initiator Device Claim allows the Initiator Device to read and write data to / from the mailbox, thus binding it to the mailbox.

* Initiator Device can also create an optional notification token for the mailbox with the Relay Server. Relay Server can notify Initiator devices when other side has deposited data in mailbox that is ready to be read. This improves user experience over polling mechanism that the devices would have to use otherwise.

* Initiator Device calls CreateMailbox API endpoint on a Relay server and provides Device Claim and optional Notification token. Relay server creates the mailbox and assigns a unique Mailbox Identifier generated using a good source of entropy (preferably hardware-based entropy).

* A mailbox has limited lifetime configured with mandatory "expiration" parameter in mailboxConfiguration. When expired, the mailbox SHALL be deleted - refer to DeleteMailbox endpoint. Relay server SHALL be responsible to periodically check for mailboxes that are past the expiration time and delete them.

* Relay server builds a unique URL link to a mailbox (for example, https://relayserver.example.com/v1/m/1234567890) and returns it to the Initiator Device.

* Initiator sends this link as invitation to Recipient Device over communication method preferred by users.

* Recipient Device, having obtained both the URL link and the Secret, is ready to read the mailbox upon user action.

* Recipient Device generates a unique token - a Recipient Device Claim - to be sent to Relay Server. Recipient Device Claim allows the Recipient Device to read and write data to / from the mailbox, thus binding it to the mailbox.

* Recipient Device can also create an optional notification token for the mailbox with the Relay Server for snappy user experience.

* Recipient Device calls ReadSecureContentFromMailbox API endpoint on the Relay Server and provides Device Claim and optional Notification token. If this is the first Recipient claim, Relay server allows the read and binds the device to the mailbox. Thus establishing a connection between Initiator and Recipient devices facilitated by Relay Server.

* Initiator Device or bound Recipient Device may delete the mailbox using the DeleteMailbox API call.

"

"

Initiator Device
Relay Server
Recipient Device
Initiator User
Recipient User
Share this Credential with Recipient User

over communication method m_1

>

Create and encrypt Provisioning

Info message_1 encrypted with Secret

CreateMailbox

(With DeviceClaim and Noti
fication token)

>

URL link to mai
lbox

<

URL link a
nd Secret

over pref
erred communication method m_1

>

Accept the Credential

<

ReadSecureContentFromMailbox

(With DeviceClaim)

<

encrypted info

>

Decrypt with Secret to get Provisioning Info message_1

Generate Provision Info message_2

encrypted with Secret

UpdateMailbox(encrypted info)

<

OK

>

Push Notificat
ion

<

ReadSecureContentFro
mMailbox

>

encrypted inf
o

<

Decrypt with Secret to get Provision Info message_2

Update with Provision Info message_3

encrypted with Secret

UpdateMailbox(encryp
ted info)

>

OK

<

Push Notification

>

ReadSecureContentFromMailbox

<

encrypted info

>

Decrypt with Secret for Provision Info message_3

DeleteMailbox

<

```
                             OK

                ›



                             Finish Credential Provisioning
      Initiator User
   Recipient User
                                            Initiator Device
      Relay Server               Recipient Device
            "
                       "
```

5. HTTP Headers

5.1. Mailbox-Request-ID

All requests to and from Relay server will have an HTTP header "Mailbox-Request-ID". The corresponding response to the API will have the same HTTP header, which SHALL echo the value in the request header. This is used to identify the request associated to the response for a particular API request and response pair. The value SHOULD be a UUID [RFC4122]. The request originator SHALL match the value of this header in the response with the one sent in the request. If response is not received, caller may retry sending the request with the same value of "Mailbox-Request-ID". Relay server SHOULD store the value of the last successfully processed "Mailbox-Request-ID" for each device based on the caller's Device Claim. A key-value pair of "Device Claim" to "Mailbox-Request-ID" is suggested to store the last successfully processed request for each device. In case of receiving a request with duplicated "Mailbox-Request-ID", Relay SHOULD respond to the caller with status code 201, ignoring the duplicate request body content.

5.2. Mailbox-Device-Claim

All requests to CreateMailbox, ReadSecureContentFromMailbox and UpdateMailbox endpoints MUST contain this header. The value represents "Device Claim" (refer to Terminology)

5.3. Mailbox-Device-Attestation

Request to CreateMailbox MAY contain this header. The value represents a Device Attestation (String, Optional) - optional remote OEM device proprietary attestation data

6. HTTP access methods

6.1. CreateMailbox

An application running on a remote device can invoke this API on Relay Server to create a mailbox and store secure data content to it (encrypted data specific to a provisioning partner). MailboxIdentifier is created by the Relay server as an UUID [RFC4122], using cryptographic entropy. A URL to the created mailbox to be returned to the caller in the response.

6.1.1. Endpoint

POST /{version}/m

6.1.2. Request Parameters:

Path parameters

* version (String, Required) - the version of the API. At the time of writing this document, v1.

Header parameters

* Mailbox-Device-Attestation (String, Optional) - optional remote OEM device proprietary attestation data.

* Mailbox-Device-Claim (String, UUID, Required) - Device Claim (refer to Terminology).

* Mailbox-Request-ID (String, UUID, Required) - Unique request identifier.

6.1.3. Consumes

This API call consumes the following media types via the Content-Type request header: application/json

6.1.4. Produces

This API call produces the following media types via the Content-Type response header: application/json

6.1.5. Request body

Request body is a complex structure, including the following fields:

* payload (Object, Required) - for the purposes of Tigress API, this is a data structure, describing Provisioning Information specific to Credential Provider. It consists of the following 2 key-value pairs:

  1. "type": "AEAD_AES_128_GCM" (refer to Encryption Format section).

  2. "data": BASE64-encoded binary value of ciphertext.

* displayInformation (Object, Required) - for the purposes of the Tigress API, this is a data structure. It allows an application running on a receiving device to build a visual representation of the credential to show to user. The data structure contains the following fields:

  1. title (String, Required) - the title of the credential (e.g. "Car Key")

  2. description (String, Required) - a brief description of the credential (e.g. "a key to my personal car")

  3. imageURL (String, Required) - a link to a picture representing the credential visually.

* notificationToken (Object, Optional) - optional notification token used to notify an appropriate remote device that the mailbox data has been updated. Data structure includes the following (if notificationToken is provided it should include both fields):

  1. type (String, Required) - notification token name. Used to define which Push Notification System to be used to notify appropriate remote device of a mailbox data update. (E.g. "com.apple.apns" for APNS)

  2. tokenData (String, Required) - notification token data (data encoded based on specific device OEM notification service rules - e.g. HEX-encoded or Base64-encoded) - application-specific - refer to appropriate Push Notification System specification.

* mailboxConfiguration (Object, Optional) - optional mailbox configuration, defines access rights to the mailbox, mailbox expiration time. Required at the time of the mailbox creation. OEM device may provide this data in the request, Relay server shall define a default configuration, if it is not provided in the incoming request. Data structure includes the following:

1. accessRights (String, Optional) - optional access rights to the mailbox for Initiator and Recipient devices. Default access to the mailbox is Read and Delete. Value is defined as a combination of the following values: "R" - for read access, "W" - for write access, "D" - for delete access. Example" "RD" - allows to read from the mailbox and delete it.

2. expiration (String, Required) - Mailbox expiration time in "YYYY-MM-DDThh:mm:ssZ" format (UTC time zone) [RFC3339]. Mailbox has limited lifetime. Once expired, it SHALL be deleted - refer to DeleteMailbox endpoint. Relay server SHOULD periodically check for expired mailboxes and delete them.


```
{
   "notificationToken": {
        "type":"com.apple.apns",
        "tokenData":"APNS1234...QW"
    }
}
```


Figure 1: Apple Push Token Example


```
{
    "displayInformation" : {
        "title" : "Hotel Pass",
        "description" : "Some Hotel Pass",
        "imageURL" : "https://example.com/sharingImage"
    },
    "payload" : {
        "type": "AEAD_AES_128_GCM",
        "data": "FDEC...987654321"
    },
    "notificationToken" : {
        "type" : "com.apple.apns",
        "tokenData" : APNS...1234"
    },
    "mailboxConfiguration" : {
        "accessRights" : "RWD",
        "expiration" : "2022-02-08T14:57:22Z"
    }
}
```


Figure 2: Create Mailbox Request Example

6.1.6. Responses

200 Status: 200 (OK)

ResponseBody:

* urlLink (String, Required) - a full URL link to the mailbox including fully qualified domain name and mailbox Identifier. Refer to "Share URL" section for details.

* isPushNotificationSupported (boolean, Required) - indicates whether push notification is supported or not. The device uses this field to decide whether it should listen on the push topic or do long-polling.


```
{
    "urlLink":"https://relayserver.example.com/m/12345678-9...A-BCD",
    "isPushNotificationSupported":true
}
```


Figure 3: Create Mailbox Response Example

201 Status: 201 (Created) - response to a duplicated request (duplicated "Mailbox-Request-ID"). Relay server SHALL respond to duplicated requests with 201 without creating a new mailbox. "Mailbox-Request-ID" passed in the first CreateMailbox request's header SHOULD be stored by the Relay server and compared to the same value in the subsequent requests to identify duplicated requests. If duplicate is found, Relay SHALL not create a new mailbox, but respond with 201 instead. The value of "Mailbox-Request-ID" of the last successfully completed request SHOULD be stored based on the Device Claim passed by the caller.

400 Bad Request - invalid request has been passed (can not parse or required fields missing).

401 Unauthorized - calling device is not authorized to create a mailbox. E.g. a device presented an invalid device claim or device attestation.

6.2. UpdateMailbox

An application running on a remote device can invoke this API on Relay Server to update secure data content in an existing mailbox (encrypted data specific to a Provisioning Partner). The update effectively overwrites the secure payload previously stored in the mailbox.

6.2.1. Endpoint

PUT /{version}/m/{mailboxIdentifier}

6.2.2. Request Parameters

Path parameters:

* version (String, Required) - the version of the API. At the time of writing this document, v1.

* mailboxIdentifier(String, Required) - MailboxIdentifier (refer to Terminology).

Header parameters:

* Mailbox-Device-Attestation (String, Optional) - optional remote OEM device proprietary attestation data.

* Mailbox-Device-Claim (String, UUID, Required) - Device Claim (refer to Terminology).

* Mailbox-Request-ID (String, UUID, Required) - Unique request identifier.

6.2.3. Consumes

This API call consumes the following media types via the Content-Type request header: application/json

6.2.4. Produces

This API call produces following media types via the Content-Type request header: application/json

6.2.5. Request body

Request body is a complex structure, including the following fields:

* payload (Object, Required) - for the purposes of Tigress API, this is a data structure, describing Provisioning Information specific to Credential Provider. It consists of the following 2 key-value pairs:

  1. "type": "AEAD_AES_128_GCM" (refer to Encryption Format section).

  2. "data": BASE64-encoded binary value of ciphertext.

* notificationToken (Object, Optional) - optional notification token used to notify an appropriate remote device that the mailbox data has been updated. Data structure includes the following (if notificationToken is provided it should include both fields):

   1. type (String, Required) - notification token name. Used to define which Push Notification System to be used to notify appropriate remote device of a mailbox data update. (E.g. "com.apple.apns" for APNS)

   2. tokenData (String, Required) - notification token data (data encoded based on specific device OEM notification service rules - e.g. HEX-encoded or Base64-encoded) - application-specific - refer to appropriate Push Notification System specification.


```
{
     "payload" : {
        "type": "AEAD_AES_128_GCM",
        "data": "FDEC...987654321"
    },
    "notificationToken":{
        "type" : "com.apple.apns",
        "tokenData" : APNS...1234"
    }
}
```


Figure 4: Update Mailbox Request Example

6.2.6. Responses

ResponseBody:

* isPushNotificationSupported (boolean, Required) - indicates whether push notification is supported or not. The device uses this field to decide whether it should listen on the push topic or do long-polling.


```
{
    "isPushNotificationSupported":true
}
```


Figure 5: Update Mailbox Response Example

200 Status: 200 (OK)

201 Status: 201 (Created) - response to a duplicate request (duplicate "Mailbox-Request-ID"). Relay server SHALL respond to duplicate requests with 201 without performing mailbox update. "Mailbox-Request-ID" passed in the first UpdateMailbox request's header SHALL be stored by the Relay server and compared to the same value in the subsequent requests to identify duplicate requests. If duplicate is found, Relay SHALL not perform mailbox update, but respond with 201 instead. The value of "Mailbox-Request-ID" of the last successfully completed request SHALL be stored based on the Device Claim passed by the caller.

400 Bad Request - invalid request has been passed (can not parse or required fields missing).

401 Unauthorized - calling device is not authorized to update the mailbox. E.g. a device presented the incorrect Device Claim.

404 Not Found - mailbox with provided mailboxIdentifier not found.

6.3. DeleteMailbox

An application running on a remote device can invoke this API on Relay Server to close the existing mailbox after it served its purpose. Recipient or Initiator Device needs to present a Device Claim in order to close the mailbox.

6.3.1. Endpoint

DELETE /{version}/m/{mailboxIdentifier}

6.3.2. Request Parameters

Path parameters:

* version (String, Required) - the version of the API. At the time of writing this document, v1.

* mailboxIdentifier(String, Required) - MailboxIdentifier (refer to Terminology).

Header parameters:

* Mailbox-Device-Claim (String, UUID, Required) - Device Claim (refer to Terminology).

* Mailbox-Request-ID (String, UUID, Required) - Unique request identifier.

6.3.3. Responses

200 Status: 200 (OK)

401 Unauthorized - calling device is not authorized to delete a mailbox. E.g. a device presented the incorrect Device Claim.

404 Not Found - mailbox with provided mailboxIdentifier not found. Relay server may respond with 404 if the Mailbox Identifier passed by the caller is invalid or mailbox has already been deleted (as a result of duplicate DeleteMailbox request).

6.4. ReadDisplayInformationFromMailbox

An application running on a remote device can invoke this API on Relay Server to retrieve public display information content from a mailbox. Display Information shall be returned in OpenGraph format (please refer to https://ogp.me for details). OpenGraph-formatted display information is required to display a preview of credential in a messaging application, e.g. iMessage or WhatsApp.

6.4.1. Endpoint

GET /{version}/m/{mailboxIdentifier}

6.4.2. Request Parameters

Path parameters:

* version (String, Required)- the version of the API. At the time of writing this document, v1.

* mailboxIdentifier(String, Required) - MailboxIdentifier (refer to Terminology).

6.4.3. Produces

This API call produces the following media types via the Content-Type response header: text/html

6.4.4. Responses

200 Status: 200 (OK)

ResponseBody :

* displayInformation (Object, Required) - visual representation of digital credential in OpenGraph format (please refer to https://ogp.me for details).

```
"<html prefix="og: https://ogp.me/ns#">
 <head>
 <title>Hotel Pass</title>
 <meta property="og:title" content="Hotel Pass" />
 <meta property="og:type" content="image/jpeg" />
 <meta property="og:description" content="Some Hotel Pass" />
 <meta property="og:url" content="share://" />
 <meta property="og:image" content="https://example.com/photos/photo.jpg" />
 <meta property="og:image:width" content="612" />
 <meta property="og:image:height" content="408" /></head>
 </html>"
```

Figure 6: Read Display Information Response Example

404 Not Found - mailbox with provided mailboxIdentifier not found.

6.5. ReadSecureContentFromMailbox

An application running on a remote device can invoke this API on Relay Server to retrieve secure payload content from a mailbox (encrypted data specific to a Provisioning Information Provider).

6.5.1. Endpoint

POST /{version}/m/{mailboxIdentifier}

6.5.2. Request Parameters

Path parameters:

* version (String, Required) - the version of the API. At the time of writing this document, v1.

* mailboxIdentifier(String, Required) - MailboxIdentifier (refer to Terminology).

Header parameters:

* Mailbox-Device-Claim (String, UUID, Required) - Device Claim (refer to Terminology).

6.5.3. Produces

This API call produces the following media types via the Content-Type response header: application/json

6.5.4. Responses

200 Status: 200 (OK)

ResponseBody :

* payload (String, Required) - for the purposes of Tigress API, this is a JSON metadata blob, describing Provisioning Information specific to Credential Provider.

* displayInformation (Object, Required) - for the purposes of the Tigress API, this is a JSON data blob. It allows an application running on a receiving device to build a visual representation of the credential to show to user. Specific to Credential Provider.

* expiration (String, Required) - the date that the mailbox will expire. The mailbox expiration time is set during mailbox creation. Expiration time should be a complete [RFC3339] date string in "YYYY-MM-DDThh:mm:ssZ" format (UTC time zone), and can be used to allow receiving clients to show when a share will expire.


```
{
    displayInformation" : {
        "title" : "Hotel Pass",
        "description" : "Some Hotel Pass",
        "imageURL" : "https://example.com/sharingImage"
    },
    "payload" : {
        "type": "AEAD_AES_128_GCM",
        "data": "FDEC...987654321"
    },
    "expiration": "2021-11-03T20:32:34Z"
}
```


Figure 7: Read Secure Content Response Example

401 Unauthorized - calling device is not authorized to read the secure content of the mailbox. E.g. a device presented the incorrect Device Claim.

404 Not Found - mailbox with provided mailboxIdentifier not found.

6.6. RelinquishMailbox

An application running on a remote device can invoke this API on Relay Server to relinquish their ownership of the mailbox. Recipient Device needs to present the currently established Recipient Device Claim in order to relinquish their ownership of the mailbox. Once relinquished, the mailbox can be bound to a different Recipient Device that presents its Device Claim in a ReadSecureContentFromMailbox call.

6.6.1. Endpoint

PATCH /{version}/m/{mailboxIdentifier}

6.6.2. Request Parameters

Path parameters:

* version (String, Required) - the version of the API. At the time of writing this document, v1.

* mailboxIdentifier(String, Required) - MailboxIdentifier (refer to Terminology).

Header parameters:

* Mailbox-Device-Claim (String, UUID, Required) - Device Claim (refer to Terminology).

* Mailbox-Request-ID (String, UUID, Required) - Unique request identifier.

6.6.3. Responses

200 Status: 200 (OK)

201 Status: 201 (Created) - response to a duplicate request (duplicate "Mailbox-Request-ID"). Relay server SHALL respond to duplicate requests with 201 without performing mailbox relinquish. "Mailbox-Request-ID" passed in the first RelinquishMailbox request's header SHALL be stored by the Relay server and compared to the same value in the subsequent requests to identify duplicate requests. If duplicate is found, Relay SHALL not perform mailbox relinquish, but respond with 201 instead. The value of "Mailbox-Request-ID" of the last successfully completed request SHALL be stored based on the Device Claim passed by the caller.

401 Unauthorized - calling device is not authorized to relinquish a mailbox. E.g. a device presented the incorrect Device Claim, or the device is not bound to the mailbox.

404 Not Found - mailbox with provided mailboxIdentifier not found. Relay server may respond with 404 if the Mailbox Identifier passed by the caller is invalid.

7. Provisioning Information Structure

The Provisioning Information is the data transferred via the Relay Server between the Initiator Device and Recipient Device. Each use case defines its own specialized Provisioning Information format, but all formats must at least adhere to the following structure. Formats are free to define new top level keys, so clients shouldn't be surprised if a message of an unexpected format has specialized top level keys.

Key	Type	Required	Description
format	String	Yes	The Provisioning Information format that the message follows. This is used by the Initiator Device and Recipient Device to know how to parse the message.
content	Dictionary	Yes	A dictionary of content to be used for the credential transfer. See each format's specification for exact fields.

Table 1

7.1. Provisioning Information Format

Each Provisioning Information format must have the message structure defined in an external specification.

Format Type	Spec Link	Description
digitalwallet.carkey.ccc	[CCC-Digital-Key-30]	A digital wallet Provisioning Information for sharing a car key that follows the Car Connectivity Consortium specification.
digitalwallet.generic.authorizationToken	[ISO-18013-5]	A digital wallet Provisioning Information for sharing a generic pass that relies solely on an authorization token.

Table 2

```
{
   "format" : "digitalwallet.carkey.ccc",
   "content": {
      // Format specific fields
   }
}
```

Figure 8: Provisioning Information format

7.2. Provisioning Information Encryption

Provisioning Information will be stored on the Relay Server encrypted. The Secret used to encrypt the Provisioning Information should be given to the Recipient Device via a "Share URL" (a URL link to a mailbox). The encrypted payload should be a data structure having the following key-value pairs:

* "type" (String, Required) - the encryption algorithm and mode used.

* "data" (String, Required) - Base64 encoded binary value of the encrypted Provisioning Information, aka the ciphertext.

Please refer to [RFC5116] for the details of the encryption algorithm.

The following algorithms and modes are mandatory to implement:

* "AEAD_AES_128_GCM": AES symmetric encryption algorithm with key length 128 bits, in GCM mode with no padding. Initialization Vector (IV) has the length of 96 bits randomly generated and tag length of 128 bits.

* "AEAD_AES_256_GCM": AES symmetric encryption algorithm with key length 256 bits, in GCM mode with no padding. Initialization Vector (IV) has the length of 96 bits randomly generated and tag length of 128 bits.

```
{
    "type" : "AEAD_AES_128_GCM",
    "data" : "IV  ciphertext  tag"
}
```

Figure 9: Secure Payload Format example

7.3. Share URL

A "Share URL" is the url a Initiator Device sends to the Recipient Device allowing it to retrieve the Provisioning Information stored on the Relay Server. A Share URL is made up of the following fields:

```
https://{RelayServerHost}/v{ApiVersion}/m/{MailboxIdentifier}?v={CredentialVertic
al}#{Secret}
```

Figure 10: Share URL example

Field	Location	Required
RelayServerHost	URL Host	Yes
ApiVersion	URI Path Parameter	Yes
MailboxIdentifier	URI Path Parameter	Yes
CredentialVertical	Query Parameter	No
Secret	Fragment	No

Table 3

7.3.1. Credential Vertical in Share URL

When a user interacts with a share URL on a Recipient Device it can be helpful to know what Credential Vertical this share is for. This is particularly important if the Recipient Device has multiple applications that can handle a share URL. For example, a Recipient Device might want to handle a general access share in their wallet app, but handle car key shares in a specific car application.

To properly route a share URL, the Initiator can include the Credential Vertical in the share URL as a query parameter. The Credential Vertical can't be included in the encrypted payload because the Recipient Device might need to open the right application before retrieving the secure payload. The Credential Vertical query parameter uses the "v" key and supports the below types. If no Credential Vertical is provided it will be assumed that this is a general access share URL.

Vertical	Value
General Access	a or _None_
Home Key	h
Car Key	c

Table 4

```
https://relayserver.example.com/v1/m/2bba630e-519b-11ec-bf63-0242ac130002?v=c#hXl
r6aRC7KgJpOLTNZaLsw==
```

Figure 11: Car Key Share URL example

The Credential Vertical query parameter can be added to the share URL by the Initiator Device when constructing the full share URL that is going to be sent to the Recipient Device.

8. Security Considerations

8.1. Relay Server defined in this document

8.1.1. Confidentiality & Integrity

* Relay Server SHALL only allow TLS connections to thwart eavesdropping or disruption of communication between Relay Server and Initiator/Recipient.

* Relay Server SHALL use Device Claim to bind Initiator and exactly one Recipient device to a mailbox. The binding prevents eavesdropping or disruption of communication between Initiator and Recipient via mailbox.

8.1.2. Network attacks

* An attacker may attempt to guess the MailboxIdentifier to eavesdrop or disrupt communication. Using version 4 UUID [RFC4122] for MailboxIdentifier SHOULD contain 122-bits of cryptographic entropy. That makes brute-force guessing attacks impractical. Also Relay Server generating MailboxIdentifiers removes any chance of collision.

* It is possible to hosting malicious or untrusted scripts by relay server preview page (ReadDisplayInformationFromMailbox). That can be mitigated by not hosting a third party JavaScripts on a preview page. Another approach is with a policy and tools to maintain the trust of such scripts (e.g. force client to verify the script against a good known hash of it).

* Relay server SHALL periodically check and delete expired mailboxes (refer to expiration parameter in the CreateMailbox request). This prevents un-authorized data leaks in future in addition to general cleanup.

8.1.3. Privacy Considerations

* Relay Server SHALL not look into data exchanged over mailbox. This is achieved by encrypting that data and making sure the Secret doesn’t land on Relay Server.

* At no time Relay server SHALL store or track the identities of Initiator and Recipient. This is achieved by letting clients pick Device Claims and Notification Tokens.

* Relay Server SHALL NOT be able to identify different mailboxes that same device is interacting with. This is achieved by letting clients pick Device Claims and Notification Tokens.

8.2. Clients of Relay Server

8.2.1. Confidentiality & Integrity

* Clients SHALL encrypt contents of the mailbox to protect it from getting revealed to the Relay Server.

* Clients SHALL check cryptographic checksum of the content to verify integrity of data. It's in the realm of Verticals to define the details and out of scope of this document.

* It is recommended that URL and secret are send separately. But if the Initiator sends both URL and the Secret as a single URL, Secret MUST be appended as URI fragment [RFC3986]. Recipient Device, upon receipt of such URL, MUST remove the Fragment (Secret) before calling the Relay server API. This ensures that Relay Server never ends up with the Secret to decode data.

```
https://relayserver.example.com/v1/m/{mailboxIdentifier}#{Secret}
```

Figure 12: Example of URL with Secret as URI Fragment

8.2.2. Privacy Considerations

* Notification Token SHALL NOT not contain identifying information. It SHOULD also be different for every new share to prevent the Relay server from correlating different shares.

* Notification token SHOULD only inform the corresponding device that there is an update available on the corresponding mailbox. Each device SHOULD keep track of all mailboxes associated with it and make read calls to appropriate mailboxes.

* The value of Mailbox-Device-Attestation header parameter SHALL not contain identifying information. It SHOULD also be different for every new share to prevent the Relay server from correlating different shares.

* Display Information is not encrypted, therefore, it SHOULD not contain any identifying information.

8.3. Overall System

The overall system security considerations are in the realm of Verticals. They are mentioned here for getting a better picture. But these are not in scope of this document as Relay Server is a piece of the overall System.

8.3.1. Initiator shares with the wrong Recipient

* Verticals allow Initiator to cancel in-flight shares and delete completed shares.

8.3.2. Malicious Recipient forwards the share to 3rd party without redeeming it or the Recipient's device is compromised.

* No mitigation, the Initiator SHOULD only share with receivers they trust.

8.3.3. Invitation Channel Security

* For better user experience, the sharing flow SHOULD allow user preferred channels. Users are familiar with these channels and use them frequently for communication. Users typically consider these channels as secure enough and trust them to deliver messages to intended recipient. Some of these channels are end to end encrypted and hence very secure and some are not. So depending on channel used it's possible that the invitation is leaked to determined attackers.

* Verticals can deploy various mitigations for this scenario.

  - Verticals SHALL ensure that the Provisioning Information of a share can only be redeemed exactly once. Relay Server helps in this by guaranteeing that only one Recipient device gets the Provisioning Information.

  - Verticals can use second factor to authenticate the Recipient. Verticals can use PIN codes, presence of Initiator Credential or other mechanisms as second factor. The second factor introduces friction to the smooth user experience during the Provisioning process or at time of use of Credential. Details of the second factor and policies around use of the second factor are out of scope of this document.

9. IANA Considerations

This document registers new headers, "Mailbox-Request-ID", "Mailbox-Device-Claim" and "Mailbox-Device-Attestation" in the "Permanent Message Header Field Names" <https://www.iana.org/assignments/message-headers>.

Header Field Name	Protocol	Status	Reference
Mailbox-Request-ID	http	std	This document
Mailbox-Device-Claim	http	std	This document
Mailbox-Device-Attestation	http	std	This document

Figure 13: Registered HTTP Header

10. References

10.1. Normative References

[CCC-Digital-Key-30]
Car Connectivity Consortium, "Digital Key Release 3", July 2022, <https://carconnectivity.org/download-digital-key-3-specification/>.

[ISO-18013-5]
Cards and security devices for personal identification, "Personal identification ISO-compliant driving license Part 5: Mobile driving license (mDL) application", September 2021, <https://www.iso.org/standard/69084.html>.

[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>.

[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>.

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, DOI 10.17487/RFC4122, July 2005, <https://www.rfc-editor.org/rfc/rfc4122>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <https://www.rfc-editor.org/rfc/rfc5116>.

[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>.

10.2. Informative References

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/RFC2818, May 2000, <https://www.rfc-editor.org/rfc/rfc2818>.

Appendix A. Contributors

The following people provided substantive contributions to this document:

* Casey Astiz
* Adam Bar-Niv
* Alexey Bulgakov
* Matt Byington
* Ben Chester
* Igor Gariev
* Manuel Gerster
* Jean-Luc Giraud
* Tommy Pauly
* Crystal Qin

Appendix B. Acknowledgments

TODO acknowledge.

Authors' Addresses

Dmitry Vinokurov
Apple Inc
Email: dvinokurov@apple.com

Yogesh Karandikar
Apple Inc
Email: ykarandikar@apple.com

Matthias Lerch
Apple Inc
Email: mlerch@apple.com

Alex Pelletier
Apple Inc
Email: a_pelletier@apple.com

Nick Sha
Alphabet Inc
Email: nicksha@google.com