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.


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

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

\[ U_i = \text{HKDF-Expand-Label}(R, "Location", Transcript, 256) \]

[[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 substituting 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:

\[
K_i = \text{HKDF-Expand-Label}(R, \text{"Key"}, \\
\quad \text{Transcript}, 256)
\]

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

```c
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]


10.2. Informative References


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