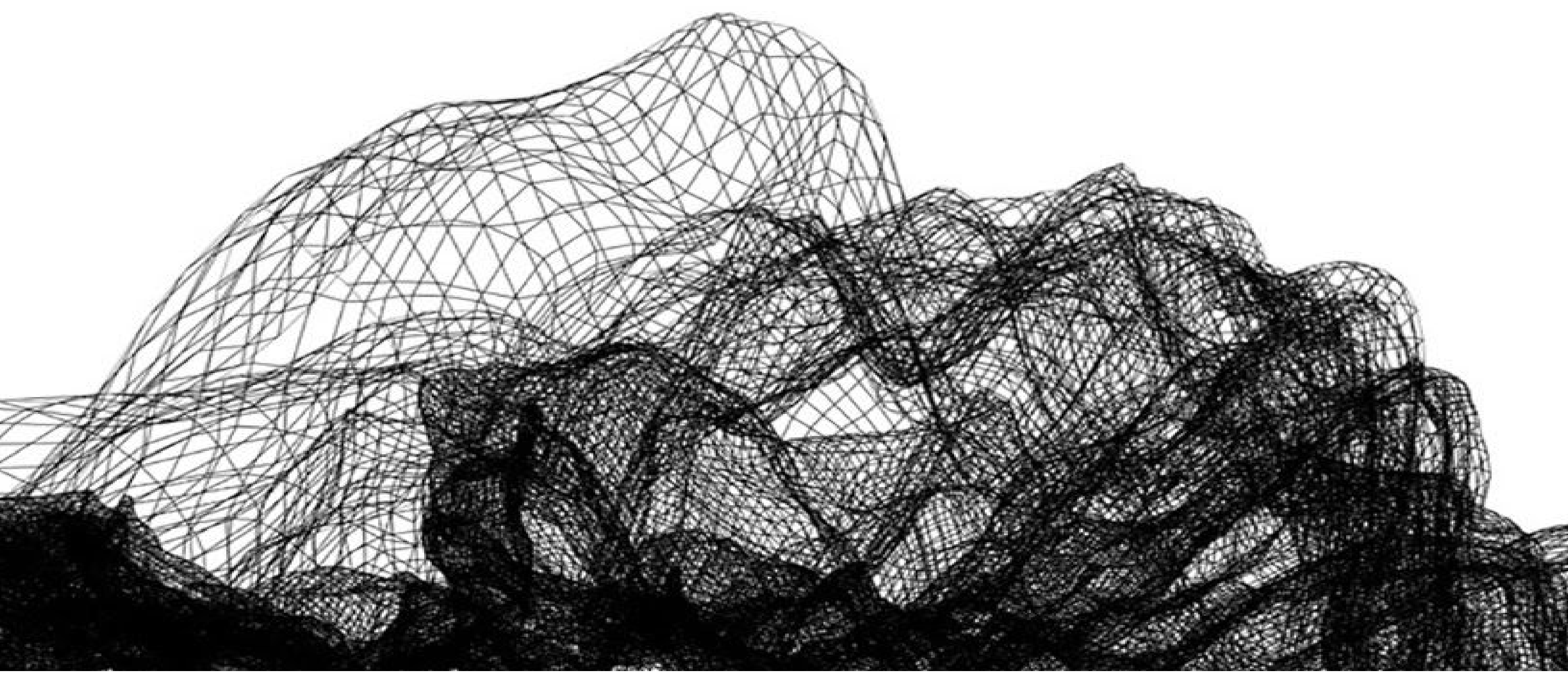
Network Transport Summary, May 2020

Date	2020-05-04
Author(s)	nmooney@duo.com, nsteele@duo.com, jerickson@duo.com
Status	EXTERNALLY PUBLISHED



Introduction

Duo Labs believes that CTAP2 should support a network-based transport. This would allow users to connect to mobile authenticator devices (such as smartphones) via the local network or the public internet.

Bluetooth is the only currently available transport that would allow a user to use their mobile device as an authenticator (barring physical connections like USB). Many desktop computers do not have Bluetooth, and not all laptops support Bluetooth Low Energy. Mobile devices also often have reason to turn off or disable bluetooth.

To increase adoption of FIDO2 authentication, we should endeavour to make it possible for all customers to fit FIDO2 into their existing deployments. A network-based transport would remove the reliance on Bluetooth and allow many more enterprises to deploy FIDO2 authentication without purchasing extra hardware.

Summary

Duo Labs proposes:

- An HTTPS-based transport, built on top of caBLE, that allows for communication with an authenticator over the public internet
- An mDNS extension to the caBLE pairing protocol allowing client-authenticator pairing between devices on the same network, without relying on Bluetooth

Communications over the public internet would be mediated by a relay server that can securely broker communications between the client and the mobile device acting as an authenticator.

caBLE

Google's proposal, caBLE v2 (which we will refer to as caBLE), is a mechanism that allows binding between a client and authenticator. A detailed description of caBLE v2 can be found in the appendix. Unlike the BLE transport, caBLE uses application-layer cryptographic binding. This allows users to sidestep the issues in the Bluetooth pairing process, and lets clients (such as Google Chrome) synchronize pairing data between multiple machines.

caBLE's pairing and communications design currently works over BLE, but the application-layer components are agnostic to the Bluetooth transport. The same pairing process, Noise handshake, and resultant Noise channel can all be implemented on top of HTTPS.

Security Considerations

We think of two stages: “pairing” and “post-pairing.” The pairing stage has certain phishing resistance requirements that are partially addressed by a proximal communication mechanism like BLE or local network communication, whereas post-pairing communications do not require the same proximity guarantees.

Proximity

Once a pairing between authenticator and client is established, a cryptographic binding between the two devices is created, and the contents of their communications are secure from eavesdropping and modification. As a result, an attacker would have to convince the user to pair the authenticator with a malicious client in order to interfere. caBLE relies on BLE’s implicit assurance of proximity to prevent remote attackers from pairing with a local device, instilling significant confidence that the user is pairing their authenticator with the intended device (i.e. their personal computer). A QR code is used to share out-of-band secrets to bootstrap an authenticator-client pairing.

Although pairing secrets are shared via the visual channel, if an attacker can phish the user by displaying a webpage that mimics the browser chrome (i.e. non-web content), the attacker may be able to convince the user to pair their mobile device with a malicious client. This is currently addressed in caBLE by the use of BLE: because the pairing happens over a range-limited channel, an attacker would also need control of Bluetooth hardware in proximity to the victim for the malicious pairing to succeed.

In a local-network scenario, two devices connected to the same local network obtain similar proximity assurances. mDNS traffic uses an address in the multicast local network control block (224.0.0/24), which is unroutable. Just as an attacker would need control of Bluetooth hardware in proximity to the victim in the BLE medium, so too would the attacker need control of a device on the local network for the malicious pairing to succeed. This similarly increases the difficulty for remote attackers to phish the pairing process.

Proximity provides some assurance of user intention during the pairing process, but does not provide any meaningful benefit once a secure channel has been established. We suggest that pairing could be completed over BLE or the local network, with future, post-pairing communications occurring over the local network or even the public Internet.

Protocol Options

We separate our local-network proposals into *pairing* and *post-pairing* stages. The security properties provided to us by authenticator-client proximity are only relevant to the pairing stage. We expect that a client that implements the network transport will implement some subset of the possible options for each stage.

Pairing

caBLE Standard BLE

This is the BLE pairing mechanism outlined in the caBLE v2 PR and described in the appendix of this document. If you are unfamiliar with the caBLE v2 PR, we recommend you jump to the appendix for context, as the following mDNS proposal is very similar.

caBLE Pairing via mDNS

In circumstances where BLE is unavailable or undesired, supporting connectivity over an alternate medium for the pairing process can provide additional options.

The core problem we address with mDNS is allowing the client and authenticator to discover each other based on the generated EID. It's important to note that our goal is simply to solve the discoverability problem. We anticipate that the actual handshake will take place via direct communication over the network using Noise, similar to what the existing caBLE proposal dictates.

Before discussing the proposed solution, it's also important to give some background on alternative solutions that might be closer to how caBLE v2 works today, showing why these won't work over mDNS:

With BLE, the authenticator can beacon arbitrarily to advertise its availability for connectivity. With mDNS, this works slightly differently. When an mDNS service is registered, the device may send an unsolicited Response message to the network, claiming a particular service name. However, RFC 6762 places strict limits on how many such announcements may be sent. Once the service has been announced, the authenticator must wait for the client to send a query message before sending another response message to the network. Consequently, the client must initiate any connections for already-paired devices. This may be an advantage over BLE in certain cases, since the FIDO2 flow begins with the client, so the authenticator does not have to be open in the foreground, beaconing periodically.

The client must start a connection by querying the network for the authenticator, but unfortunately cannot query for a particular authenticator via the EID since it is known to only one party when it is generated. In the existing caBLE v2 specification, the authenticator is responsible for generating the EID. If the client were responsible for generating the EID and embedding it into the query (for instance, <eid>._fido_c1._udp.local), the authenticator would need to perform trial decryption on all such queries and respond to any that it recognizes as corresponding to a known pairing. On mobile platforms such as Android, applications may only register services using pre-known service names, which precludes this approach.

Instead, we propose the following flow which will enable clients and authenticators to discover each other. We will cover certain portions in detail:

- The user initiates pairing on the client device
- The authenticator scans a QR code presented by the client (browser)
- With information gathered from the QR code, the authenticator creates an EID and registers an mDNS service at <EID>._fido_c1.udp.local
    - This registration should initiate an announcement of the registered service via mDNS, potentially bypassing the need for the client to send an mDNS PTR request
    - Should it be needed, the client sends an mDNS PTR request for _fido_c1._udp.local, to which all authenticators on the network are expected to respond
    - The authenticator sees this query, and returns a response containing a SRV record with the hostname (<EID>._fido_c1.udp.local) and port to which the Noise handshake should occur, as well as a TXT record that includes optional metadata
- The client, using information from the SRV response, initiates a handshake with the authenticator over the network
    - In the PairingData transmitted in the last step of the handshake, permanent pairing information (i.e. FQDN and port of a network relay) may be included
- CTAP2 is spoken over the channel established by the handshake.

Discovery via mDNS

In order for the client to discover the authenticator on the LAN after initiating the pairing, the client begins to make DNS PTR requests for _fido_c1._udp.local.

After the authenticator generates the EID, it will register the service <EID>._fido_c1._udp.local and begin listening for mDNS queries. Upon observing a query for _fido_c1._udp.local, the authenticator will create a response with the following structure:

```
<EID>._fido_c1._udp.local

SRV:
    hostname
    port
TXT:
    possible metadata
```

The client will process each observed response, attempting to decrypt the observed EID's until it records a plaintext ending in the expected 8 zero bytes.

Establishing the Handshake

Upon receipt of a DNS response containing a valid EID, the client will inspect the embedded SRV and TXT answers. The SRV record provides a hostname (or IP address) and port the client can use to establish a Noise handshake over TCP.

Due to the difficulties of holding a listening socket open in perpetuity on the mobile device, we instead propose ultimately enabling the mobile authenticator to receive a new registration/assertion flow via mobile push. This is typically done via a message broker that coordinates with the Firebase Cloud Messenger (FCM) or Apple Push Notification (APN) service to deliver a push notification to the mobile device.

To make this possible, the authenticator may specify an external FQDN and port in the PairingData (allowing the client to reach a network relay) and a network identifier (to allow the message broker to forward messages to the correct authenticator). In this case, the client knows that the information provided in the PairingData is permanent, and an mDNS discovery flow is not needed for future registrations/assertions.

If permanent pairing data is provided, the post-pairing HTTPS protocol is used.

Post-pairing

HTTPS

The client (Chrome) establishes a channel with an authenticator by connecting to the network relay:

```
POST /advertise
Headers:
```

```
    X-Cable-Network-ID: <base64(network_id)>
Body:
    <base64(eid)>
Response:
    channel_id
```

Now, with the channel ID received, the client communicates with the authenticator in request-response format, with the network relay brokering messages to/from the mobile device:

```
POST /communicate
Headers:
    X-Cable-Channel-ID: <base64(channel_id)>
Body:
    <base64(noise(ctap2))>
Response:
    <base64(noise(ctap2))>
```

The network ID in question is a global identifier that we suggest using for routing purposes, since a network relay server is likely to host multiple authenticators. This isn't a perfect solution in that it breaks the non-identifiability of the EID broadcasts, but our assumption is that the network relay needs to be sufficiently trusted to route requests to the correct authenticator anyway.

Appendix

How does caBLE v2 work?

In the existing caBLE v2 proposal, there are two steps required for an authenticator (e.g. mobile device) and a client (e.g. Chrome) to communicate: pairing and handshake. A pairing can be temporary or permanent. You may want a temporary pairing if you are using your smartphone as an authenticator with a public kiosk, whereas a permanent pairing makes sense for a personal machine.

The outline of the flow is as follows, and we will dig into each step:

- The user initiates a pairing on the client device (by clicking a “pair new device” button or similar)
- The authenticator scans a QR code
- With information gathered from the QR code, the authenticator broadcasts (via BLE) an advertisement that the client can recognize
- The client, using information from the authenticator advertisement, initiates a handshake with the authenticator
- The authenticator optionally provides permanent pairing information in the last step of the handshake
- CTAP2 is spoken over the channel established by the handshake.

Let’s dive into the specifics of how this flow works.

Pairing

- The client generates a 16-byte random QR secret, encodes it as the URI `fido://c1/<Base64(QR secret)>`, and displays the QR code
- The authenticator scans the QR code and retrieves the QR secret

Both parties are now able to calculate two keys: the EID generating key and the PSK generating key. The EID generating key is used to generate ephemeral IDs that the client will use to recognize the authenticator. The PSK generating key is used to generate PSKs that will be used to secure the handshake between client and authenticator.

```
PSK generating key =
      HKDF-SHA256(ikm = QR secret, salt = nil, info = “caBLE QR to PSK
generator key”)
```

```
EID generating key =
     HKDF-SHA256(ikm = QR secret, salt = nil, info = “caBLE QR to EID
generator key”)
```

The authenticator is now ready to advertise. It does so by generating a handshake nonce of 8 random bytes. An ephemeral ID is the result of encrypting the handshake nonce with the EID generating key.

```
ephemeral ID =
     AES-256(key = EID generating key, plaintext = handshake nonce ++
eight zero bytes)
```

The authenticator then uses BLE GATT (which we won't delve into) to beacon out the ephemeral ID. To an observer, this ephemeral ID looks randomly generated. The client that is waiting to pair with an authenticator can await these beacons and *attempt to decrypt the ephemeral ID*. If the ephemeral ID, decrypted with the EID generating key, results in a value that ends in eight zero bytes, we know we have found an authenticator that scanned the client's QR code.

The first eight bytes of the decrypted value are the handshake nonce. The nonce is then used to generate the PSK, along with the PSK generating key:

```
PSK =
     HKDF-SHA256(ikm = PSK generating key, salt = handshake nonce,
info = nil)
```

Now, the client initiates a Noise handshake with the authenticator, following the NNpsk0 pattern. The inputs to this initial handshake are the PSK generated above and the prologue string `“caBLE QR code handshake”`.

If the pairing is to be a permanent pairing, the authenticator transmits a payload alongside its final handshake message, which is a CBOR structure:

```
PairingData = {
  1 => bstr   ; EID generating key; 32-bytes
  2 => bstr   ; PSK generating key; 32-bytes
  3 => bstr   ; authenticator identity; 65-bytes; X9.62 encoded P-256
point
  4 => string ; authenticator's suggested name
}
```

These new EID generating keys and PSK generating keys are *not* the same keys as previously calculated. Instead, they are global to the authenticator. The authenticator uses global keys to

avoid having to maintain state and, in the future, transmit beacons for every pairing it has. This results in the limitation that the authenticator cannot revoke individual pairings, and must reset its global keys and invalidate all pairings if it wishes to remove one.

Post-pairing Handshake

If a pairing was previously established, a similar handshake can be conducted. No QR code is scanned. When the authenticator app is opened on the mobile device, the authenticator again starts beaconing an ephemeral ID. Again, the ephemeral ID is:

```
ephemeral ID =
      AES-256(key = EID generating key, plaintext = handshake nonce ++
eight zero bytes)
```

The EID generating key used here is the global key transmitted in the PairingData payload above. The handshake nonce is again eight random bytes.

The client listens for beacons, and performs *trial decryption* on each beacon it sees, using each EID generating key it knows (one per pairing). Once it finds eight zero bytes as a payload suffix, it can again calculate a PSK:

```
PSK =
      HKDF-SHA256(ikm = PSK generating key, salt = handshake nonce,
info = nil)
```

The client uses this PSK, the prologue string "`caBLE handshake`", and the authenticator identity from the PairingData as input to the NKpsk0 Noise handshake.

Limitations

- As mentioned above, the EID and PSK generating keys are global to an authenticator, so individual pairings cannot be revoked
- The authenticator must initiate the handshake process by beaconing. The caBLE v2 PR comments that the client could initiate the handshake by choosing a handshake nonce and beaconing in a similar fashion, with a different suffix pattern (say, eight bytes of 0x01). This is not supported by caBLE v2 at the moment because the BLE stacks of clients are not sufficiently well-behaved to support this across platforms.

Perhaps the largest limitation of this scheme is that it relies on BLE. This is by design, but we can already see that some limitations (such as the authenticator-led handshake process) are caused by BLE. An mDNS channel for caBLE v2 messages may alleviate some of these issues while retaining the proximity requirements that led to BLE's use.

DUO
LABS