Proximity Exchange Protocol

Working Draft, July 17, 2026

This version:
https://fidoalliance.org/specs/hybrid/proximity-exchange-protocol-v1.0-wd-20260717.html

Previous Versions:

Issue Tracking:
GitHub

Editor:
David Waite (Ping Identity)

Contributors:
Tim Cappalli (Okta)
Harsh Lal (Google)

Abstract

This specification describes a method for establishing a communication channel for leveraging a credential store on one device from other devices, establishing proximity and user intent.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current FIDO Alliance publications and the latest revision of this technical report can be found in the FIDO Alliance specifications index at https://fidoalliance.org/specifications/.

This document was published by the FIDO Alliance as a Working Draft Specification. If you wish to make comments regarding this document, please Contact Us. All comments are welcome.

This is a Working Draft Specification and is not intended to be a basis for any implementations as the Specification may change. This document is merely a FIDO Alliance working group internal and **member-confidential** document. It has no official standing of any kind and does not represent consensus of the FIDO Alliance. No rights are granted to prepare derivative works of this Specification. Entities seeking permission to reproduce portions of this Specification for other uses must contact the FIDO Alliance to determine whether an appropriate license for such use is available.

Implementation of certain elements of this Specification may require licenses under third party intellectual property rights, including without limitation, patent rights. The FIDO Alliance, Inc. and its Members and any other contributors to the Specification are not, and shall not be held, responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

THIS FIDO ALLIANCE SPECIFICATION IS PROVIDED “AS IS” AND WITHOUT ANY WARRANTY OF ANY KIND, INCLUDING, WITHOUT LIMITATION, ANY EXPRESS OR IMPLIED WARRANTY OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Table of Contents

1. Introduction§

This section is not normative.

The Proximity Exchange Protocol (PXP, formerly CTAP hybrid transport) decouples the proof that theclient platform is physically close to the authenticator orcredential manager hosting device (CMHD) from the transport of messages (CTAP2, JSON etc.) between them. The protocol is intended to connect credential-hosting devices with cameras, typically phones, to a client platform.

PXP involves a data transfer channel and proof of device proximity. Bluetooth Low Energy (BLE) advertisements are used for proof of proximity. The data transfer channel can either leverage network communication via a service called a tunnel service, or use local communication (e.g. Bluetooth Low Energy (BLE), Ultra-wideband (UWB), etc.). A **tunnel service** is a highly available network service with a domain name known to the CMHD that use it.

1.1. Relationship to Other Specifications§

The Proximity Exchange Protocol was created as a transport mechanism to allow for cross-device (e.g. cross-ecosystem) authentication for CTAP 2. It has since expanded to support cross-device usage of Digital Credentials API.

2. Overview§

This section is not normative.

PXP operates in three stages - invocation, channel negotiation and communication.

The client platform initiates creation of a channel by advertising its intent and capabilities, typically as a response to a user action such as requesting cross-domain authentication.

Initiation is typically done by displaying a QR code, but also may rely on state established previously with a Credential Manager Hosting Device. At the same time, theclient platform will begin listening for the CMHD to make a BLE advertisement based on the information in the QR code.

The CMHD will perform a BLE advertisement to start the negotiation stage. The negotiation stage is where proximity is verified, end to end encryption between the two parties is established, and any channel configuration is exchanged.

Once a channel is established, the CMHD will convey additional capabilities before encrypted messages are exchanged.

3. Conformance§

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this specification are to be interpreted as described in [RFC2119].

CMHDs and Client Platforms may implement additional constraints on these specifications to meet the certification requirements of programs like [CMVP], [CSPN], and [CommonCriteria].

4. Terminology§

Client Platform
: Expands upon the Web Authentication usage of the term

Credential Manager Hosting Device (CMHD)
: In the context of PXP, the CMHD is the device running the credential manager software which is remotely accessed by the client platform. For FIDO2 credentials, the credential manager acts as a roaming authenticator. For identity credentials (e.g. verifiable or digital credentials), the credential manager acts as an identity wallet.

5. Invocation and Channel Negotiation§

This section describes different mechanisms through which a data transfer channel can be established between a client platform and an authenticator devices. Devices MAY support multiple invocation mechanisms.

5.1. FIDO URI Scheme§

A FIDO URI is used to initiate an interaction between two devices. It is a scheme optimized for efficient encoding into QR codes.

The URI is written in the form "FIDO:/", followed by digit-encoded CBOR, henceforth referred to as**qr data**. The scheme is always written in uppercase to optimize data storage within a QR code. A single forward-slash MUST follow the colon due to limitations in URI recognition in some QR-capable camera software. A double forward-slash MUST NOT be used as that would indicate an authority, which this URI scheme does not use.

Digit-encoding is designed to be efficient when expressed in a QR code. Seven-byte chunks are interpreted as little-endian values and encoded as 17-digit, base 10 numbers. Any remaining bytes are encoded likewise using the minimum number of digits that some value of that number of bytes could need. Specifically, since the remainder is known to be 1, 2, 3, 4, 5, or 6 bytes long, its encoded form will take 3, 5, 8, 10, 13, or 15 digits, respectively.

```
func digitEncode(d []byte) string {
  const chunkSize = 7
  const chunkDigits = 17
  const zeros = "00000000000000000"

  var ret string
  for len(d) >= chunkSize {
    var chunk [8]byte
    copy(chunk[:], d[:chunkSize])
    v := strconv.FormatUint(binary.LittleEndian.Uint64(chunk[:]), 10)
    ret += zeros[:chunkDigits-len(v)]
    ret += v

    d = d[chunkSize:]
  }

  if len(d) != 0 {
    // partialChunkDigits is the number of digits needed to encode
    // each length of trailing data from 6 bytes down to zero. I.e.
    // it's 15, 13, 10, 8, 5, 3, 0 written in hex.
    const partialChunkDigits = 0x0fda8530

    digits := 15 & (partialChunkDigits >> (4 * len(d)))
    var chunk [8]byte
    copy(chunk[:], d)
    v := strconv.FormatUint(binary.LittleEndian.Uint64(chunk[:]), 10)
    ret += zeros[:digits-len(v)]
    ret += v
  }

  return ret
}
```

The encoded data is a CBOR map with integer keys mapping to key-specific values. The CBOR MUST be in canonical form.

5.2. CBOR Message Encoding§

Many transports (e.g., Bluetooth Smart) are bandwidth-constrained, and serialization formats such as JSON are too heavy-weight for such environments. For this reason, all encoding is done using the concise binary encoding CBOR [RFC8949].

To reduce the complexity of the messages and the resources required to parse and validate them, all messages

MUST use the CTAP2 canonical CBOR encoding form as specified below, which differs from the "Deterministically Encoded CBOR" suggested in Section 4.2 of [RFC8949]. All encoders MUST serialize CBOR in the CTAP2 canonical CBOR encoding form without duplicate map keys. All decoders SHOULD reject CBOR that is not validly encoded in the CTAP2 canonical CBOR encoding form and SHOULD reject messages with duplicate map keys.

The **CTAP2 canonical CBOR encoding form** uses the following rules:

- Integers MUST be encoded as small as possible.
  - 0 to 23 and -1 to -24 MUST be expressed in the same byte as the major type;
  - 24 to 255 and -25 to -256 MUST be expressed only with an additional uint8_t;
  - 256 to 65535 and -257 to -65536 MUST be expressed only with an additional uint16_t;
  - 65536 to 4294967295 and -65537 to -4294967296 MUST be expressed only with an additional uint32_t.
- The representations of any floating-point values are not changed.

> NOTE: The size of a floating point value—16-, 32-, or 64-bits—is considered part of the value for the purpose of CTAP2. E.g., a 16-bit value of 1.5, say, has different semantic meaning than a 32-bit value of 1.5, and both can be canonical for their own meanings.

- The expression of lengths in major types 2 through 5 MUST be as short as possible. The rules for these lengths follow the above rule for integers.
- Indefinite-length items MUST be made into definite-length items.
- The keys in every map MUST be sorted lowest value to highest. The sorting rules are:
  - If the major types are different, the one with the lower value in numerical order sorts earlier.
  - If two keys have different lengths, the shorter one sorts earlier;
  - If two keys have the same length, the one with the lower value in (byte-wise) lexical order sorts earlier.

> NOTE: These rules are equivalent to a lexicographical comparison of the canonical encoding of keys for major types 0-3 and 7 (integers, strings, and simple values). They differ for major types 4-6 (arrays, maps, and tags), which CTAP2 does not use as keys in maps. These rules should be revisited if CTAP2 does start using the complex major types as keys.

- Tags as defined in Section 3.4 in[RFC8949] MUST NOT be present.

Because some processors are memory constrained, the depth of nested CBOR structures used by all message encodings is limited to at most four (4) levels of any combination of CBOR maps and/or CBOR arrays. CMHDs MUST support at least 4 levels of CBOR nesting. Clients, platforms, and servers MUST NOT use more than 4 levels of CBOR nesting.

Likewise, because some processors are memory constrained, the maximum message size supported by an CMHD MAY be limited. By default, CMHDs MUST support messages of at least 1024 bytes. CMHDs MAY declare a different maximum message size supported using the maxMsgSize authenticatorGetInfo result parameter. Clients, platforms, and servers MUST NOT send messages larger than 1024 bytes unless the CMHD's maxMsgSize indicates support for the larger message size. CMHD MAY return the CTAP2_ERR_REQUEST_TOO_LARGE error if size or memory constraints are exceeded.

If map keys are present that an implementation does not understand, they MUST be ignored. Note that this enables additional fields to be used as new features are added without breaking existing implementations.

Messages from the host to the CMHD are called "commands" and messages from CMHD to host are called "responses". All values are big endian encoded.

CMHDs SHOULD return the CTAP2_ERR_INVALID_CBOR error if received CBOR does not conform to the requirements above.

Several commands reference externally-defined structures (such as `PublicKeyCredentialRpEntity` from Web Authentication) which, for the purposes of this protocol, are encoded as CBOR. The rules and behaviours for processing such CBOR are defined above, but such structures can also be invalid because of missing required fields, or because values have an incorrect type. If structures in messages from the host are missing required members, or the values of those members have the wrong type, then the CMHD SHOULD return CTAP2_ERR_CBOR_UNEXPECTED_TYPE.

5.3. Initiation

This section describes different mechanisms through which a data transfer channel can be established between a client platform and an authenticator devices. Devices MAY support one or more initiation mechanisms.

5.3.1. QR-initiated Transactions

When the client platform wishes to communicate with a CMHD, it may display a QR code that contains a public key and a shared secret key. The public key authenticates the client platform to any connecting CMHD and knowledge of the secret key authenticates the connecting CMHD to the client platform.

```
var (
  qrSecret [16]byte
  // The ecdsa package is used for its convenient public/private key
  // structures, but these are ECDH keys, not ECDSA.
  identityKey *ecdsa.PrivateKey
)

func showQRCode() {
  rand.Reader.Read(qrSecret[:])

  var err error
  identityKey, err = ecdsa.GenerateKey(elliptic.P256(), rand.Reader)
  if err != nil {
    panic(err)
  }
  identityKeyCompressed := compressECKey(&identityKey.PublicKey)

  printQRCode(encodeQRContents(&identityKeyCompressed, &qrSecret))
}
```

The contents of the QR code are a FIDO URI, with the following CBOR keys:

Key 0
: a 33-byte, P-256, X9.62, compressed public key.

Key 1
: a 16-byte random QR secret.

Key 2
: the number of assigned tunnel server domains known to this implementation (see `decodeTunnelServerDomain` for details).

Key 3
: (optional) the current time in epoch seconds.

Key 4
: (optional) a boolean that is true if the device displaying the QR code can perform state-assisted transactions.

Key 5
: a value from the table below, representing the user flow to follow. Implementations SHOULD treat unknown values as `ga`. This field exists so that guidance can be given to the user immediately upon scanning the QR code, prior to the CMHD receiving any CTAP message or JSON request. While this hint SHOULD be as

accurate as possible, it does not constrain the subsequent CTAP messages or JSON requests that the platform may send.

Value	Description
`ga`	getAssertion (FIDO2)
`mc`	makeCredential (FIDO2)
`dcp`	credential presentation (Digital Credentials API)
`dci`	credential issuance (Digital Credentials API)

Key 6

(optional) a list of integers denoting transport channels supported by the client. If this value is not present, it is assumed to be a list with a single element corresponding to WebSockets for backwards compatibility.

Value	Description
0	WebSockets
1	Bluetooth Low Energy

CMHDs MUST use a CBOR parser to parse this information, as more keys MAY be added in the future. Implementations MUST ignore unknown keys.

Implementations (such as the function below) MAY use GREASE (e.g.[rfc8701]) to encourage implementations to support extensibility. Integer keys below -256 or above 255 are reserved for such usage.

```
func encodeQRContents(compressedPublicKey *[33]byte, qrSecret *[16]byte) string {
  numMapElements := 7
  // GREASE QR code to ensure that keys can be added later.
  var randByte [1]byte
  rand.Reader.Read(randByte[:])
  extraKey := randByte[0]&3 == 0
  if extraKey {
    numMapElements++
  }

  var cbor []byte
  cbor = append(cbor, 0xa0+byte(numMapElements))       // CBOR map
  cbor = append(cbor, 0)                               // key 0
  cbor = append(cbor, (cborMajorByteString<<5)|24, 33) // 33 bytes
  cbor = append(cbor, compressedPublicKey[:]...)
  cbor = append(cbor, 1)                               // key 1
  cbor = append(cbor, (cborMajorByteString<<5)|16) // 16 bytes
  cbor = append(cbor, qrSecret[:]...)

  cbor = append(cbor, 2) // key 2
  n := len(assignedTunnelServerDomains)
  if n > 24 {
    panic("larger encoding needed")
  }
  cbor = append(cbor, byte(n))

  cbor = append(cbor, 3) // key 3
  cbor = append(cbor, cborEncodeInt64(time.Now().Unix())...)

  cbor = append(cbor, 4) // key 4
  cbor = append(cbor, 0xf5)  // true

  cbor = append(cbor, 5) // key 5
  cbor = append(cbor, (cborMajorByteString<<5)|2, 'm', 'c')

  cbor = append(cbor, 6)                     // key 6
  cbor = append(cbor, (cborMajorArray<<5)|2) // array of 2 elements
  cbor = append(cbor, 0)                     // first element of array
  cbor = append(cbor, 1)                     // second element of array

  if extraKey {
    cbor = append(cbor, 0x19, 0xff, 0xff, 0) // key 65535, value 0
  }

  qr := "FIDO:/" + digitEncode(cbor)
  fmt.Println(qr)
  return qr
}
```

5.3.2. State-assisted Transactions§

If a client platform has linking information for a CMHD, from a previous QR-initiated transaction, then it doesn't need to show a QR code in order to contact that CMHD again. By making a WebSockets connection to the cached tunnel service with the path `/cable/contact/` followed by the base64url-encoded contact ID, the tunnel service will attempt to establish a tunnel with the identified CMHD. If the tunnel service believes that the CMHD is permanently uncontactable (e.g. because the user opted to unlink this client platform on the CMHD) then the tunnel server returns HTTP status 410 and the client platform should forget the link information.

The CMHD needs two values to start communicating on the tunnel: the link ID so that it knows which client

platform is contacting it (and thus which keys to use), and a nonce from theclient platform. The latter diversifies the key that encrypts the BLE advert and prevents anyone passively listening from being able to link the advert to any set of link keys retrospectively. The two values are called the “client payload” and are hex-encoded in a `X-caBLE-Client-Payload` HTTP header.

In order to aid theCMHD in displaying UI to the user, a third value is encoded in the client payload: a hint about whether the following transaction will be a makeCredential or a getAssertion.

Once the tunnel is ready theCMHD will send its handshake message and start advertising over BLE as a proximity challenge. The BLE advert in this case contains the same initial flags byte, which must be zero, and the remaining 15 bytes are all nonce. Once the BLE advert is received, the client platform can calculate the handshake PSK and respond.

The handshake in this case will beNKpsk0 because now it is theCMHD that has previously shared a public key.

```
func performStateAssistedConnection(linkData *linkData) {
  contactURL := "wss://" +
    linkData.tunnelServerDomain +
    "/cable/contact/" +
    base64.RawURLEncoding.EncodeToString(linkData.ContactID)

  clientNonce, clientPayload := constructClientPayload(linkData)
  headers := make(http.Header)
  headers.Add("X-caBLE-Client-Payload", hex.EncodeToString(clientPayload))

  websocketConn, resp, err := (&websocket.Dialer{
    Subprotocols: []string{subprotocol},
  }).Dial(contactURL, headers)

  if err != nil {
    if resp != nil && resp.StatusCode == 410 {
      panic("device unlinked")
    }
    panic(err)
  }

  if websocketConn.Subprotocol() != subprotocol {
    panic("tunnel service picked wrong subprotocol")
  }

  var eidKey [64]byte
  derive(eidKey[:], linkData.LinkSecret[:], clientNonce[:], keyPurposeEIDKey)

  println("waiting for advert")
  advertPlaintext := awaitAdvert(eidKey)
  println("have advert")
  if !reservedBitsAreZero(advertPlaintext) {
    panic("bad link advert")
  }

  var psk [32]byte
  derive(psk[:], linkData.LinkSecret[:], advertPlaintext[:], keyPurposePSK)

  doHandshake(websocketConn, psk, nil, linkData.authPublicKey)
  println("State-assisted connection complete")
}
```

The client payload is encoded in a CBOR message (which must follow theencoding rules) using the following format:

- Key 1: the 8-byte link ID; a bytestring.

- Key 2: a 16-byte nonce generated by the client platform; a bytestring.
- Key 3: a value from the table below, representing the user flow to follow.

Value	Description
`ga`	getAssertion (FIDO2)
`mc`	makeCredential (FIDO2)
`dcp`	credential presentation (Digital Credentials)
`dci`	credential issuance (Digital Credentials)

```
func constructClientPayload(linkData *linkData) (nonce [16]byte, payload []byte) {
  rand.Reader.Read(nonce[:])

  payload = append(payload, 0xa3)                    // Three-element CBOR map
  payload = append(payload, 1)                       // key 1
  payload = append(payload, cborMajorByteString<<5|8) // 8 bytes
  payload = append(payload, linkData.LinkID[:]...)
  payload = append(payload, 2)                        // key 2
  payload = append(payload, cborMajorByteString<<5|16) // 16 bytes
  payload = append(payload, nonce[:]...)
  payload = append(payload, 3)                    // key 3
  payload = append(payload, cborMajorString<<5|2) // two-byte string
  payload = append(payload, 'g', 'a')             // getAssertion

  return nonce, payload
}
```

From this point, the connection works the same as the QR-initiated one. The CMHD can optionally send linking information in the post-handshake message if it wishes to update any linking information and then CTAP2 messages flow as before.

5.3.3. NFC-initiated Transactions§

If the client platform and the authenticator both support Near Field Communication (NFC), then the flow can be initiated over NFC by tapping the authenticator device to the client platform device.

On detecting the NFC tag, the client platform sends a `SELECT` APDU command to the authenticator device.

The FIDO2 AID for NFC invocations consists of the following fields:

Field	Value
`RID`	0xA000000647
`PIX`	0x2F0002

The command structure to send a `SELECT` APDU command is:

CLA	INS	P1	P2	Lc	Data In	Le
0x00	0xA4	0x04	0x00	Length of AID	AID	Variable

A successful `SELECT` allows the client platform to know that the applet is present and active. A client MUST send this command to the authenticator before any other command. In response to the applet selection command, the

authenticator returns status word SW_OK SW1(90) SW2(00).

On receiving SW_OK, the client sends the PUT DATA APDU command with qr data to the authenticator. On receiving the PUT DATA command, the authenticator extracts qr data from the command and uses it to connect to the client platform post proximity checks via one of the data transfer channels described later.

The command structure to send a PUT DATA APDU command is:

CLA	INS	P1	P2	Lc	Data In	Le
0x00	0xDA	0x01	0x11	Length of qr data	qr data	Variable

In response to the PUT DATA APDU command, the authenticator SHALL return SW_OK status word SW1(90) SW2(00).

The authenticator SHOULD extract the qr data from the command and use it to connect to the client platform post proximity checks via one of the data transfer channels described later.

The authenticator may return SW_UNKNOWN_COMMAND status word SW1(6D) SW2(00) in case an unknown APDU command is received.

5.4. Proximity

The client platform awaits a connection attempt from an authenticator. PXP requires a proof of proximity to help prevent attacks, thus notification of the connection attempt comes in the form of a BLE advertisement. (Without a proof of proximity a web site could, for example, display a QR code and attempt to convince the user to scan it with their authenticator. By having the authenticator demand that the client platform prove reception of a BLE advert such an attacker would have to have control of a Bluetooth radio near to the victim.)

5.4.1. Bluetooth Low Energy Advertisement

Once the QR code has been displayed, the client platform awaits a connection attempt from an CMHD. This transport requires a proof of proximity to help prevent attacks, thus notification of the connection attempt comes in the form of a BLE advertisement. (Without a proof of proximity a web site could, for example, display a QR code and attempt to convince the user to scan it with their CMHD. By having the CMHD demand that the client platform prove reception of a BLE advert such an attacker would have to have control of a Bluetooth radio near to the victim.)

The UUID 0000fff9-0000-1000-8000-00805f9b34fb must be included in the advert, and client platforms must require that candidate devices are advertising this UUID. That UUID must also have a 20-byte service data payload, which is trial decrypted to search for a match to the displayed QR code. The size of the payload may be larger if the advertisement suffix is appended to it. The Bluetooth extended advertising capability is required to support the advertisement suffix.

The **advertisement suffix** is a CBOR map containing extra information the data channel needs for establishing a connection, serialized to bytes. This SHOULD not be sent if empty.

The format for the advertisement suffix is defined as:

```
<suffix:CborMap> = {
  <transport_channel_identifier:CborInteger> : <channel_extra:CborValue>
}
```

transport_channel_identifier is the integer identifier of the selected transport channel, as defined by Key 6 of the QR code CBOR map.

`channel_extra` is a CBOR value specific to the channel. In this version of the specification, this is only used for the Bluetooth Low Energy channel.

```
func awaitAdvert(eidKey [64]byte) [16]byte {
  // uuidsChan is a channel of UUID sets observed from some BLE device.
  // Each UUID is represented as a string in the standard format, e.g.
  // 0000fde2-0000-1000-8000-00805f9b34fb.
  var (
    serviceDataChan chan map[string][]byte
    stopScanning    func()
    err             error
  )

  if serviceDataChan, stopScanning, err = bleScanForServiceData(); err != nil {
    panic(err)
  }
  defer stopScanning()

  const UUID = "0000fff9-0000-1000-8000-00805f9b34fb"

  for serviceData := range serviceDataChan {
    cableData, ok := serviceData\[UUID]
    if !ok {
      continue
    }

    // Only first 20 bytes are decrypted. The advertisement suffix is parsed separately.
    if payload, ok := trialDecrypt(&eidKey, cableData[:20]); ok {
      return payload
    }
  }

  panic("UUID channel closed")
}
```

If the client platform does not include BLE as one of the supported transport channels, then the CMHD parses the first 20-bytes of data and discards the rest if it exists. Otherwise, the service data can be parsed as follows:

1. Parse first 20-bytes of service data as described in later sections.
2. Additional bytes are parsed as an advertisement suffix.

In order to derive the key needed to trial decrypt BLE adverts, the following key derivation is used. Whenever a key is needed for a specific purpose it is always derived from a parent key in order to ensure domain separation. The derivation uses [RFC5869] with SHA-256, where the input keying material is the parent key, the salt is an optional input, and the info value is a 32-bit, little-endian, purpose identifier.

```
type keyPurpose uint32

const (
  keyPurposeEIDKey   keyPurpose = 1
  keyPurposeTunnelID keyPurpose = 2
  keyPurposePSK      keyPurpose = 3
)

func derive(output, secret, salt []byte, purpose keyPurpose) {
  if uint32(purpose) >= 0x100 {
    panic("unsupported purpose")
  }

  var purpose32 [4]byte
  purpose32[0] = byte(purpose)

  h := hkdf.New(sha256.New, secret, salt, purpose32[:])
  if n, err := h.Read(output); err != nil || n != len(output) {
    panic("HKDF error")
  }
}
```

The key used to decrypt adverts is then a 64-byte value derived from the QR secret with `keyPurposeEIDKey`. The term “EID” is historical and does not stand for anything here.

```
func awaitQRAdvert() [16]byte {
  var eidKey [32 + 32]byte
  derive(eidKey[:], qrSecret[:], nil, keyPurposeEIDKey)
  return awaitAdvert(eidKey)
}
```

When decrypting adverts, these 64 bytes of EID key are considered as a pair of 256-bit keys where the first 32 bytes are an AES key and the second 32 bytes are an HMAC-SHA256 key. A candidate BLE advert is valid if the final four bytes are a correct HMAC tag of the other 16 bytes. For each valid BLE advert, those initial 16 bytes are then taken to be an AES block and decrypted with the AES key.

This is a poor-man’s substitute for a wide-block mode, but wide-block modes are non-standard. There is no more space in the BLE advert so a nonce cannot be included. Since it’s possible that two CMHDs could scan the same QR code and broadcast based on the same key, avoiding a mode that XORs plaintext with a keystream avoids potential complications.

```
func trialDecrypt(eidKey *[64]byte, candidateAdvert []byte) (plaintext [16]byte, ok bool) {
  var zeros [16]byte
  if len(candidateAdvert) != 20 {
    return zeros, false
  }

  h := hmac.New(sha256.New, eidKey[32:])
  h.Write(candidateAdvert[:16])
  expectedTag := h.Sum(nil)

  if !hmac.Equal(expectedTag[:4], candidateAdvert[16:]) {
    return zeros, false
  }

  block, err := aes.NewCipher(eidKey[:32])
  if err != nil {
    panic(err)
  }

  block.Decrypt(plaintext[:], candidateAdvert[:16])
  if !reservedBitsAreZero(plaintext) {
    return zeros, false
  }

  return plaintext, true
}
```

Once successfully authenticated and decrypted, a BLE advert yields 16 bytes of plaintext. These 16 bytes consist of (in order):

- A flags byte, which is currently zero. This could be used for versioning in the future.
- 80 bits of connection nonce.
- A 24-bit routing ID.
- A 16-bit tunnel service identifier.

```
func reservedBitsAreZero(plaintext [16]byte) bool {
  return plaintext[0] == 0
}

func unpackDecryptedAdvert(plaintext [16]byte) (
  nonce [10]byte,
  routingID [3]byte,
  encodedTunnelServerDomain uint16) {

  copy(nonce[:], plaintext[1:])
  copy(routingID[:], plaintext[11:])
  encodedTunnelServerDomain = uint16(plaintext[14]) | (uint16(plaintext[15]) << 8)
  return
}
```

The connection nonce is the value that demonstrates possession of the BLE advert, and thus proximity to the CMHD.

6. Communications§

After the CMHD and platform verify proximity and exchange encryption keys, the devices negotiate a data transfer channel over which to communicate. More details on how to negotiate and send data over the various

data transfer channels and the communication protocol within those channels are described below.

6.1. Data transfer channel§

PXP supports multiple data transfer channels. The client platform advertises the supported channels in the QR code. The CMHD device maintains a set of its supported channels. On scanning the QR code, it finds the intersection between the two sets to find a set of common supported data transfer channels. The CMHD is responsible for determining which available data transport channel is most appropriate to use for communication, and MAY attempt start a connection through more than one data transfer channels from the intersection.

Once a connection is established through a data transfer channel, the CMHD SHOULD discard any other attempted channels. Additional channels in PXP are provided to help improve reliability in certain environments, such as ones without a network connection. Data transfer over multiple channels concurrently is not supported. The WebSocket data transfer channel SHOULD be supported by both client platform and CMHD for fallback and backwards compatibility.

6.1.1. WebSockets§

The tunnel service relays messages to and from the CMHD. It is a property of the CMHD because, as detailed later, it can contact the CMHD on request when a client platform is “linked”. The protocol between the CMHD and the tunnel service, and details about how the service later contacts the CMHD, are a private detail of the CMHD’s implementation.

The encoded tunnel service identifier is a `uint16`. Values `0` through `255` are assigned, and values over `255` are translated into a domain name by hashing. A “`cable`” label is prepended to hashed domains to allow for use of CNAME records.

Domains are assigned sequentially and the number of assigned domains is included in the QR code. Therefore CMHDs can know whether a peer will recognize an assigned domain or not and can potentially fall back to a hashed domain for compatibility.

These are the currently assigned domains, in order:

```
var assignedTunnelServerDomains = []string{"cable.ua5v.com", "cable.auth.com"}

func decodeTunnelServerDomain(encoded uint16) (string, bool) {
  if encoded < 256 {
    if int(encoded) >= len(assignedTunnelServerDomains) {
      return "", false
    }
    return assignedTunnelServerDomains[encoded], true
  }

  shaInput := []byte{
    0x63, 0x61, 0x42, 0x4c, 0x45, 0x76, 0x32, 0x20,
    0x74, 0x75, 0x6e, 0x6e, 0x65, 0x6c, 0x20, 0x73,
    0x65, 0x72, 0x76, 0x65, 0x72, 0x20, 0x64, 0x6f,
    0x6d, 0x61, 0x69, 0x6e,
  }
  shaInput = append(shaInput, byte(encoded), byte(encoded>>8), 0)
  digest := sha256.Sum256(shaInput)

  v := binary.LittleEndian.Uint64(digest[:8])
  tldIndex := uint(v & 3)
  v >>= 2

  ret := "cable."
  const base32Chars = "abcdefghijklmnopqrstuvwxyz234567"
  for v != 0 {
    ret += string(base32Chars[v&31])
    v >>= 5
  }

  tlds := []string{".com", ".org", ".net", ".info"}
  ret += tlds[tldIndex&3]

  return ret, true
}
```

The routing ID is an opaque value that must be provided to the tunnel service and which aids its operation.

The client platform is now in possession of everything needed to establish the tunnel to theCMHD. The first step of doing so is to derive the tunnel ID, a 128-bit identifier that the tunnel service recognizes and which identifies the exchange separate from any others that the tunnel service might concurrently be facilitating. It is derived, as detailed above, from the QR secret. It is not dependent on the nonce from the BLE advert because that would mean that the tunnel service could try and brute-force the nonce from the tunnel ID. The tunnel service is trusted by the CMHD, but no need to trust it more than necessary.

With the tunnel ID in hand, the tunnel service is contacted viaWebSockets. In order to request a connection to a given tunnel ID, the path of the WebSockets URL is set to `/cable/connect/` followed by the lower-case, hex-encoded routing ID, another foreslash, then the lower-case, hex-encoded tunnel ID. The WebSocket connection must set the subprotocol identifier to `fido.cable`.

Implementations must follow HTTP redirects when establishing the WebSocket connection.

```
var tunnelServerDomain string

const subprotocol = "fido.cable"

func connectToPhone(advertPlaintext [16]byte) {
  _, routingID, encodedTunnelServerDomain := unpackDecryptedAdvert(advertPlaintext)

  var ok bool
  if tunnelServerDomain, ok = decodeTunnelServerDomain(encodedTunnelServerDomain); !ok {
    panic("unknown tunnel server domain")
  }

  var tunnelID [16]byte
  derive(tunnelID[:], qrSecret[:], nil, keyPurposeTunnelID)

  connectURL := "wss://" +
    tunnelServerDomain +
    "/cable/connect/" +
    hex.EncodeToString(routingID[:]) +
    "/" +
    hex.EncodeToString(tunnelID[:])

  conn, _, err := (&websocket.Dialer{
    Subprotocols: []string{subprotocol},
  }).Dial(connectURL, nil)

  if err != nil {
    panic(err)
  }

  if conn.Subprotocol() != subprotocol {
    panic("tunnel service picked wrong subprotocol")
  }

  doQRHandshake(conn, advertPlaintext)
}
```

With the tunnel established, messages are exchanged in binary WebSocket frames and no other frame types are permitted on the connection.

6.1.2. Bluetooth Low Energy

Similar to the WebSocket data transfer channel, a Bluetooth Low Energy (BLE) data transfer channel can be established as well. The CMHD, after processing the contents of the QR code, checks if BLE is amongst one of the supported transport channels by the client platform. If supported, the CMHD may choose to create an insecure L2CAP| Connection-oriented Channel (CoC) Bluetooth server socket. This socket can be used to listen for incoming connections. A PSM value (henceforth called server psm) uniquely identifying this channel is auto-generated. The CMHD then adds the server psm into the advertisement suffix under the corresponding key, signalling the client that it can accepting BLE L2CAP connections. A **server psm** is an integer value denoting Protocol/Service Multiplexer for L2CAP channel.

The BLE `transport_channel_identifier` for advertisement suffix is 1. The BLE `channel_extra` is a Cbor integer whose value is the server PSM

The client parses the server PSM (if it exists) from the advertisement data, which can be used to connect to the insecure Bluetooth L2CAP Connection-oriented Channel (CoC) socket. This channel can be used for further message transfers. Every message transmitted over this socket MUST be prefixed with a 4-byte big-endian length prefix, representing the length of the message payload in bytes. Bluetooth Low Energy L2CAP and extend

ed advertising are optional features and may not be available on all devices. Implementations SHOULD always include websockets as a fallback.

6.2. Data Transfer§

The CMHD and client platform first perform a cryptographic handshake to establish a forward-secure, authenticated connection. This handshake is Noise KNpsk0 using P-256, SHA-256, and AES-256-GCM.

The client platform speaks first to prove possession of the BLE advert. TheCMHD thus needs only to receive the client platform's handshake message and send a reply in order to complete the handshake. TheKNpsk0 pattern requires that the initiator (the client platform) have shared a public key in advance with the responder (the CMHD), and that both sides share a symmetric key. The pre-exchanged public key was passed to theCMHD in the QR code, and the pre-shared symmetric key is derived from the QR secret and decrypted BLE advert. (The full BLE advert is included in the PSK derivation to ensure that any future additions to the advert format are automatically authenticated.)

```
func doQRHandshake(socketConn *socket.Conn, advertPlaintext [16]byte) {
  var psk [32]byte
  derive(psk[:], qrSecret[:], advertPlaintext[:], keyPurposePSK)

  conn, handshakeHash := doHandshake(socketConn, psk, identityKey, nil)
  readPostHandshakeMessage(conn, handshakeHash)
}

func doHandshake(socketConn *socket.Conn,
  psk [32]byte,
  identityKey *ecdsa.PrivateKey,
  // peerIdentity is not used until linked connections are discussed, below.
  peerIdentity *ecdsa.PublicKey) (

  conn io.ReadWriteCloser,
  handshakeHash [32]byte) {

  msg, ephemeralKey, noiseState := initialHandshakeMessage(psk, identityKey, peerIdentity)
  if err := socketConn.WriteMessage(socket.BinaryMessage, msg); err != nil {
    panic(err)
  }

  msgType, handshakeMessageFromPhone, err := socketConn.ReadMessage()
  if err != nil {
    panic(err)
  }
  if msgType != socket.BinaryMessage {
    panic("non-binary message received on Socket")
  }

  trafficKeys, handshakeHash := processHandshakeResponse(
    handshakeMessageFromPhone, ephemeralKey, identityKey, noiseState)

  conn = newCableConn(&socketAdaptor{socketConn}, trafficKeys)
  return conn, handshakeHash
}
```

As referenced above, the handshake itself isNoise NKpsk0. The following functions implement bothNKpsk0 and KNpsk0 because the latter will be needed below. The underlying Noise operations are specified inthe Noise specification. p256X962Length is the length of an uncompressed, X9.62, P-256 point, in bytes.

```
const p256X962Length = 1 + 32 + 32
```

```
func initialHandshakeMessage(
  psk [32]byte,
  priv *ecdsa.PrivateKey,
  peerPub *ecdsa.PublicKey) (

  msg []byte,
  ephemeralKey *ecdsa.PrivateKey,
  noise *noiseState) {

  if (priv == nil) == (peerPub == nil) {
    panic("exactly one of priv and peerPub must be given")
  }

  var ns *noiseState
  if peerPub != nil {
    ns = newNoise(noiseNKpsk0)
    ns.mixHash([]byte{0})
    ns.mixHashPoint(peerPub)
  } else {
    ns = newNoise(noiseKNpsk0)
    ns.mixHash([]byte{1})
    ns.mixHashPoint(&priv.PublicKey)
  }

  ns.mixKeyAndHash(psk[:])

  ephemeralKey, err := ecdsa.GenerateKey(elliptic.P256(), rand.Reader)
  if err != nil {
    panic(err)
  }

  ephemeralKeyBytes := elliptic.Marshal(ephemeralKey.Curve, ephemeralKey.X, ephemeralKey.Y)
  ns.mixHash(ephemeralKeyBytes)
  ns.mixKey(ephemeralKeyBytes)

  if peerPub != nil {
    ns.mixKey(ecdh(ephemeralKey, peerPub.X, peerPub.Y))
  }

  msg = append(msg, ephemeralKeyBytes...)
  msg = append(msg, ns.encryptAndHash(nil)...)

  return msg, ephemeralKey, ns
}

func processHandshakeResponse(
  peerHandshakeMessage []byte,
  ephemeralKey *ecdsa.PrivateKey,
  priv *ecdsa.PrivateKey,
  ns *noiseState) (

  keys trafficKeys,
  handshakeHash [32]byte) {

  if len(peerHandshakeMessage) < p256X962Length {
    panic("handshake too short")
  }

  peerPointBytes := peerHandshakeMessage[:p256X962Length]
  ciphertext := peerHandshakeMessage[p256X962Length:]

  ns.mixHash(peerPointBytes)
  ns.mixKey(peerPointBytes)
```

```
  ns.mixKey(peerPointBytes)

  peerPointX, peerPointY := elliptic.Unmarshal(ephemeralKey.Curve, peerPointBytes)
  if peerPointX == nil {
    panic("peer's point is not on the curve")
  }

  ns.mixKey(ecdh(ephemeralKey, peerPointX, peerPointY))

  if priv != nil {
    ns.mixKey(ecdh(priv, peerPointX, peerPointY))
  }

  plaintext, ok := ns.decryptAndHash(ciphertext)
  if !ok || len(plaintext) != 0 {
    panic("bad handshake")
  }

  return ns.split(), ns.handshakeHash()
}
```

Once the handshake is complete, the traffic-keys that result from Noise's `Split` operation are assigned to the client platform-to-CMHD and CMHD-to-client platform flows, respectively. Future messages on the tunnel are padded and AES-256-GCM encrypted. Padding is performed by setting the final byte of the plaintext to the number of preceding bytes that are padding. Padding bytes can take any value but zero is recommended. Implementations can use a padding granularity up to 256 bytes, but 32 is recommended. Nonces are per-direction counters, big-endian encoded into 12 bytes. The additional data for every message is empty.

Implementations may terminate connections that exceed 24 bits of nonce to avoid worrying about nonce overflow.

```
type cableConn struct {
  conn              io.ReadWriteCloser
  readKey, writeKey [32]byte
  readSeq, writeSeq uint32
}

var additionalData []byte = nil

func setupAEAD(counter *uint32, key *[32]byte) (nonce [12]byte, aead cipher.AEAD) {
  if *counter > 1<<24 {
    // To avoid dealing with the nonce counter overflowing,
    // connections are capped at 2^24 messages.
    panic("too many messages")
  }

  binary.BigEndian.PutUint32(nonce[8:], *counter)
  *counter++

  block, err := aes.NewCipher(key[:])
  if err != nil {
    panic(err)
  }
  if aead, err = cipher.NewGCM(block); err != nil {
    panic(err)
  }

  return
}

func (c *cableConn) Write(msg []byte) (int, error) {
  const paddingGranularity = 32
  if len(msg) > 1<<20 {
    // 1MiB is comfortably larger than any valid CTAP2 message and
    // this limit moots possible overflows below.
    panic("plaintext too large")
  }

  extraBytes := paddingGranularity - (len(msg) % paddingGranularity)
  paddedLen := len(msg) + extraBytes

  paddedMsg := make([]byte, paddedLen, paddedLen)
  copy(paddedMsg, msg)
  paddedMsg[len(paddedMsg)-1] = byte(extraBytes) - 1

  nonce, aead := setupAEAD(&c.writeSeq, &c.writeKey)
  ciphertext := aead.Seal(paddedMsg[:0], nonce[:], paddedMsg, additionalData)

  if n, err := c.conn.Write(ciphertext); err != nil {
    return 0, err
  } else if n != len(ciphertext) {
    return 0, errors.New("unexpected short write")
  } else {
    return len(msg), nil
  }
}
```

Decryption consists of the reverse of the encryption steps:

```
func (c *cableConn) Read(buf []byte) (int, error) {
  n, err := c.conn.Read(buf)
  if err != nil {
    return n, err
  }
  buf = buf[:n]

  nonce, aead := setupAEAD(&c.readSeq, &c.readKey)
  plaintext, err := aead.Open(buf[:0], nonce[:], buf, additionalData)
  if err != nil {
    panic("decryption failure")
  }

  if len(plaintext) == 0 {
    panic("invalid message")
  }
  paddingBytes := int(plaintext[len(plaintext)-1])
  if paddingBytes+1 > len(plaintext) {
    panic("invalid message")
  }

  plaintext = plaintext[:len(plaintext)-1-paddingBytes]
  if len(plaintext) > len(buf) {
    panic("message too large")
  }
  n = copy(buf, plaintext)
  return n, nil
}
```

The first message from the CMHD is the “post handshake” message. This message contains the CMHD’s getInfo response, to save a round-trip. This message contains a CBOR map, which must be in CTAP2 canonical form.

The CBOR map contains the following:

- Key 0: (optional) a bytestring containing only zero bytes, for padding.
- Key 1: the getInfo response, a bytestring.
- Key 2: reserved.
- Key 3: (optional) an array of strings representing supported features, as defined in the table below. The absense of this key MUST be treated as if it were present with the value ["ctap"].

Value	Description
`ctap`	The credential manager hosting device (CMHD) supports CTAP2 requests.
`dc`	The credential manager hosting device (CMHD) supports Digital Credentials requests using JSON-based messages.

```
type postHandshakeMessage struct {
  GetInfoReply []byte   \`cbor:"1"\`
}

func readPostHandshakeMessage(conn io.ReadWriteCloser, handshakeHash [32]byte) {
  msgBytes := make([]byte, 128<<10)
  n, err := conn.Read(msgBytes)
  if err != nil {
    panic("read failure: " + err.Error())
  }

  var msg postHandshakeMessage
  if !cborParse(&msg, msgBytes[:n]) {
    fmt.Printf("%x\n", msgBytes)
    panic("invalid post-handshake message")
  }

  if msg.GetInfoReply == nil {
    panic("post-handshake message is missing getInfo response")
  }

  sendCTAP2Request(conn, handshakeHash)
}
```

With the tunnel now fully set up, the parties can exchange messages. Each message begins with a byte that denotes the type of the message. An empty message is thus a protocol error. The following types are defined:

- 0: a shutdown message.
- 1: a CTAP message.
- 2: an update message.
- 3: a JSON-based message.

A shutdown message may only be sent by the client platform to the CMHD. The message must consist only of the type byte. It indicates that the client will not send any further messages to the CMHD. The CMHD may choose to close the connection upon receiving such a message. If it supports state-assisted transactions then the client platform SHOULD accept messages from the CMHD for at least two minutes after sending a shutdown message.

A CTAP message contains a CTAP2 payload for processing. For example, when sent from client platform to CMHD, the bytes following the type byte will be one of the CTAP2 commands.

An update message may be sent by either side at any time. The bytes following the type byte must be a CBOR map encoded using the canonical rules. Unknown keys in the map must be ignored. The codespace of keys is separate for each direction. Currently keys are only defined in the CMHD to client platform direction:

- Key 0: (optional) a bytestring containing only zero bytes, for padding.
- Key 1: (optional) a map containing linking information.

The linking map contains:

- Key 1: the “contact ID”, an opaque value that can be presented to the tunnel service to identify thisCMHD. (For Android this an FCM registration token.)
- Key 2: the “link ID”, an opaque value that identifies this link to theCMHD. This must be sent back to the CMHD when contacting it so that it knows what set of keys to use for thisclient platform.
- Key 3: the “link secret”, a shared secret key.
- Key 4: the CMHD’s public key, X9.62 uncompressed. This value is global to theCMHD and identifies it. If the same CMHD is used multiple times with a a QR-initiated transaction then this lets theclient platform

deduplicate the linking information. Desktops may sync linking information using systems like Chrome Sync and this public key prevents a client platform with linking information from impersonating theCMHD to another client platform.

- Key 5: the CMHD's name, for the purposes of identifying it to the user. For example "Pixel 3 XL".
- Key 6: the handshake signature. See below.

```
type authenticatorToClientUpdateMessage struct {
  LinkingData linkData   \`cbor:"1"\`
}

type linkData struct {
  ContactID              []byte   \`cbor:"1"\`
  LinkID                 [8]byte  \`cbor:"2"\`
  LinkSecret             [32]byte \`cbor:"3"\`
  AuthenticatorPublicKey [65]byte \`cbor:"4"\`
  AuthenticatorName      string   \`cbor:"5"\`
  Signature              [32]byte \`cbor:"6"\`

  authPublicKey      *ecdsa.PublicKey
  tunnelServerDomain string
}

var initialLinkData *linkData

func parseUpdateMessage(payload []byte, handshakeHash [32]byte) {
  var msg authenticatorToClientUpdateMessage
  if !cborParse(&msg, payload) {
    fmt.Printf("%x\n", payload)
    panic("invalid update message")
  }

  // Linking data is optional.
  if msg.LinkingData.ContactID == nil {
    return
  }

  initialLinkData = &msg.LinkingData

  pubKey := &ecdsa.PublicKey{
    Curve: elliptic.P256(),
  }
  pubKey.X, pubKey.Y = elliptic.Unmarshal(pubKey.Curve, initialLinkData.AuthenticatorPublicK
ey[:])
  if pubKey.X == nil {
    panic("bad link public key")
  }

  if !verifySignature(initialLinkData.Signature, handshakeHash, pubKey) {
    panic("invalid link signature")
  }

  initialLinkData.tunnelServerDomain = tunnelServerDomain
  initialLinkData.authPublicKey = pubKey

  fmt.Printf("Linking information received\n")
}
```

The signature in the linking data serves to prove possession of the claimed public key. This is needed because that public key is an identifier and future linking messages that claim the same public key will replace older ones.

This allows a CMHD to update its linking information at theclient platform, but CMHDs should not be able to replace another CMHD's data.

The handshake hash is Noise's channel binding value and hashes the handshake transcript. Since theCMHD's public key is used as an ECDH key in later Nosie handshakes, we don't want to overload it as an ECDSA key too. Thus the "signature" in the linking message is actually an HMAC of the handshake hash under the shared key between the CMHD's key and the key in theclient platform's QR code.

```
func verifySignature(sig, handshakeHash [32]byte, pubKey *ecdsa.PublicKey) bool {
  sharedKey := ecdh(identityKey, pubKey.X, pubKey.Y)
  h := hmac.New(sha256.New, sharedKey)
  h.Write(handshakeHash[:])
  expectedSignature := h.Sum(nil)
  return hmac.Equal(expectedSignature, sig[:])
}
```

The client platform must send CTAP2 commands in order to direct the CMHD to perform some action. Typically in a CTAP2 exchange that would be a getInfo request. However, since the response was already provided in the post-handshake message, the client platform can immediately send a more substantial request. The example below sends a superfluous `authenticatorGetInfo` request.

```
const (
  typeShutdown = 0
  typeCTAP = 1
  typeUpdate = 2
  typeJSON = 3
)

func sendCTAP2Request(conn io.ReadWriteCloser, handshakeHash [32]byte) {
  authenticatorGetInfoRequest := []byte{typeCTAP, 4}
  if _, err := conn.Write(authenticatorGetInfoRequest); err != nil {
    panic("write failed")
  }

  for {
    reply := make([]byte, 128<<10)
    n, err := conn.Read(reply)
    if err != nil {
      fmt.Printf("Socket closed\n");
      return
    }
    reply = reply[:n]

    if len(reply) == 0 {
      panic("invalid empty message received")
    }

    msgType, reply := reply[0], reply[1:]

    switch msgType {
    case typeShutdown:
      panic("shutdown message received from authenticator")

    case typeCTAP:
      fmt.Printf("CTAP reply: %x\n", reply)
      if _, err := conn.Write([]byte{typeShutdown}); err != nil {
        panic("write failed")
      }

    case typeUpdate:
      parseUpdateMessage(reply, handshakeHash)

    default:
      panic("invalid message type received")
    }
  }

  conn.Close()
}
```

7. JSON-based Messages§

7.1. Feature detection§

Support for JSON-based messages, and more specifically Digital Credentials API requests using JSON-based messages, is determined by the presence of the string `dc` in the array for key 3 of the post handshake message's CBOR map as defined in QR-Initiated Transactions.

7.2. Request Properties§

The following JSON schema defines the properties of a JSON-based request:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://schemas.fidoalliance.org/ctap/json-request/v2_2.schema.json",
  "title": "JSON-based Request",
  "type": "object",
  "properties": {
    "origin": {
      "type": "string",
      "description": "The caller's origin as determined by the client platform."
    },
    "requestType": {
      "type": "string",
      "description": "The type of request.",
      "anyOf": [
        {
          "const": "credential.get",
          "description": "A get request from Credential Management or the app platform equivale
nt."
        },
        {
          "const": "credential.create",
          "description": "A create request from Credential Management or the app platform equiv
alent."
        }
      ]
    },
    "request": {
      "type": "object",
      "description": "One or more requests of the same requestType.",
      "properties": {
        "digital": {
          "type": "object",
          "description": "The [=Digital Credentials API=] request object."
        }
      }
    }
  },
  "additionalProperties": false,
  "required": [
    "origin",
    "requestType",
    "request"
  ]
}
```

EXAMPLE 1

JSON-based request for a digital credential

```
{
  "origin": "https://verify1.example.com",
  "requestType": "credential.get",
  "request": {
    "digital": {
      "protocol": "openid4vp-v1-unsigned",
      "data": {
        "response_type": "vp_token",
        "response_mode": "dc_api",
        "nonce": "GqTvQNhCCFszZORB8MXJ0lGqQOP3EhBQ7veOe6j-1Kk",
        "dcql_query": {
          "credentials": [
            {
              "id": "age_proof",
              "format": "mso_mdoc",
              "meta": {
                "doctype_value": "org.iso.18013.5.1.mDL"
              },
              "claims": [
                {"path": ["org.iso.18013.5.1", "age_over_21"]},
                {"path": ["org.iso.18013.5.1", "family_name"]},
                {"path": ["org.iso.18013.5.1", "given_name"]}
              ]
            }
          ]
        }
      }
    }
  }
}
```

7.3. Response Properties

The following JSON schema defines the properties of a JSON-based response:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://schemas.fidoalliance.org/ctap/json-response/v2_2.schema.json",
  "title": "JSON-based Response",
  "type": "object",
  "properties": {
    "response": {
      "type": "object",
      "description": "One or more responses matching the request.",
      "properties": {
        "digital": {
          "type": "object",
          "description": "A response to a Digital Credential request.",
          "maxProperties": 1,
          "properties": {
            "data": {
              "type": "object",
              "description": "The [=Digital Credentials API=] response object."
            },
            "error": {
              "type": "string",
              "description": "For an unsuccessful ceremony, the error code describing the error
dition.",
              "anyOf": [
                {
                  "const": "USER_CANCELLED",
                  "description": "The user actively cancelled the request."
                },
                {
                  "const": "DEVICE_ABORTED",
                  "description": "The device aborted the request."
                },
                {
                  "const": "NO_CREDENTIAL",
                  "description": "No credential found to satisfy the request."
                }
              ]
            },
            "additionalProperties": false
          }
        }
      }
    }
  },
  "additionalProperties": false,
  "required": [
    "response"
  ]
}
```

EXAMPLE 2

Successful JSON-based response for a digital credential:


```
{
  "response": {
    "digital": {
      "data": {
        "protocol": "openid4vp-v1-unsigned",
        "data": {
          "vp_token": {
            "age_proof": "o2d2ZXJzaW9uYzEuMGlkb2N1bWVudHOBo2dkb2NUeXBldW9yZy5pc28uMTgwMTMuN
S4xLm1ETGxpc3N1ZXJTaWduZWSiam5hbWVTcGFjZXOhcW9yZy5pc28uMTgwMTMuNS4xg9gYWFSkaGRpZ2VzdElEAGZy
W5kb21QpGwYPdo8unC7_AZnK-693HFlbGVtZW50SWRlbnRpZmllcmtmYW1pbHlfbmFtZWxlbGVtZW50VmFsdWVlU21p
GjYGFhRpGhkaWdlc3RJRAFmcmFuZG9tUBDBlc8u3nJsVKpG8VUKc-hxZWxlbWVudElkZW50aWZpZXJqZ2l2ZW5fbmFt
WxlbGVtZW50VmFsdWVjSm9u2BhYT6RoZGlnZXN0SUQCZnJhbmRvbVA83Kw692RRua44-G_JeysAcWVsZW1lbnRJZGVu
GlmaWVya2FnZV9vdmVyXzIxbGVsZW1lbnRWYWx1ZfVqaXNzdWVyQXV0aIRDoQEmoRghWQJLMIICRzCCAe2gAwIBAgIU
K0P_dVwQ5WfFEhbntqgZ9oynsUwCgYIKoZIzj0EAwIweTELMAkGA1UEBhMCVVMxEzARBgNVBAgMCkNhbGlmb3JuaWEx
jAUBgNVBAcMDU1vdW50YWluIFZpZXcxHDAaBgNVBAoME0RpZ2l0YWwgQ3JlZGVudGlhbHMxHzAdBgNVBAMMFmRpZ2l0
WxjcmVkZW50aWFscy5kZXYwHhcNMjQxMTEwMDEwODAzWhcNMzQxMDI5MDEwODAzWjB5MQswCQYDVQQGEwJVUzETMBEG
1UECAwKQ2FsaWZvcm5pYTEWMBQGA1UEBwwNTW91bnRhaW4gVmlldzEcMBoGA1UECgwTRGlnaXRhbCBDcmVkZW50aWFs
zEfMB0GA1UEAwwWZGlnaXRhbGNyZWRlbnRpYWxzLmRldjBZMBMGByqGSM49AgEGCCqGSM49AwEHA0IABOtDqHrSdqvb
CsalMxtFFgsG1bJ-QfDaThDNlzjQSwEk14On5ZcrPzl0mM2WgKwLsKRvWymKvFB0pU9bLZ5EGmjUzBRMB0GA1UdDgQW
BQLLHD8AxxsbwunUTBS45pEGTnbsDAfBgNVHSMEGDAWgBQLLHD8AxxsbwunUTBS45pEGTnbsDAPBgNVHRMBAf8EBTAD
QH_MAoGCCqGSM49BAMCA0gAMEUCIQD8lbryGFFjP2Xaxy7zJbnnGLvLKrvJweDpqtMfhvvnMwIgbkMANURt0aeiHqvN
pR1cSHYSeyCMRGTq8fq7bljh8tZAbrYGFkBtaZndmVyc2lvbmMxLjBvZGlnZXN0QWxnb3JpdGhtZ1NIQS0yNTZnZG9j
HlwZXVvcmcuaXNvLjE4MDEzLjUuMS5tRExsdmFsdWVEaWdlc3RzoXFvcmcuaXNvLjE4MDEzLjUuMaMAWCD5MaTvyx9t
HER92NyQSYNJ-JVbM4YgjBOqbShB62hHwFYIBxfeIk3ALJg7LUF-X2wCO_WUS4PuJcjl0luTA9QHr5HAlggYU5Dkca2
aGsla4np5N0zY8L-UAgNHn_2lZSJGiClbBtZGV2aWNlS2V5SW5mb6FpZGV2aWNlS2V5pAECIAEhWCB07xfD2-9IWUn1
Lsl3r0D-Nnx9toOLnhf3aMhuQmbGiJYIAxMzD6Kl_nY_UykJ-qdykIqCacRdWEGu42EotRizdWFbHZhbGlkaXR5SW5m
6Nmc2lnbmVkwHgbMjAyNC0xMS0xN1QyMDo1MjoyMi45MTk3ODJaaXZhbGlkRnJvbcB4GzIwMjQtMTEtMTdUMjA6NTI6
jIuOTE5Nzg5Wmp2YWxpZFVudGlswHgbMjAzNC0xMS0wNVQyMDo1MjoyMi45MTk3ODlaWECXUu3Z3RJKxu4yLOVORV4t
vC5xqZXJSHIebo90bVbRhZcjF8sbA_SsbtmGm6iaHmSki0zuVF1H8V2Qkdc7zUxbGRldmljZVNpZ25lZKJqbmFtZVNv
WNlc9gYQaBqZGV2aWNlQXV0aKFvZGV2aWNlU2lnbmF0dXJlhEOhASag9lhAFsk536slR2rWiCCsYe2ZeBLWBDIxS7ab
oB8lAD_HA0zAzR_0rn1wZmchGgxZkUoO-vYFKVl-T28KYyJFSEfeWZzdGF0dXMA"
          }
        }
      }
    }
  }
}
```


EXAMPLE 3

Unsuccessful JSON-based response for a digital credential


```
{
  "response": {
    "digital": {
      "error": "NO_CREDENTIAL"
    }
  }
}
```


8. Related Documents

The following documents are published by other organisations and are not referenced by this specification but may be relevant to the same audience. They are gathered here purely as informational resources and are not

necessarily endorsed by FIDO.

1. Android's Credential Manager API provides a native abstraction of the WebAuthn API and also provides a mechanism for apps to claim domain names as valid RP IDs.
2. Apple provides a native abstraction of the WebAuthn API and provides a mechanism for apps to claim domain names as valid RP IDs.
3. Windows provides a native abstraction of the WebAuthn API to applications.

9. IANA Considerations§

9.1. Uniform Resource Identifier (URI) Schemes Registration§

This section registers the URI Scheme value defined in Section § 5.1 FIDO URI Scheme in the IANA "Uniform Resource Identifier (URI) Schemes" registry [IANA-URI-Schemes] established by [RFC7595].

Scheme name
fido

Status
permanent

Applications/protocols that use this scheme name
Specifications from the FIDO Alliance (https://fidoalliance.org/) for cross-device communication.

Contact
David Waite <dwaite@pingidentity.com>

Change controller
FIDO Alliance

References
Section § 5.1 FIDO URI Scheme of this document.

10. Security Considerations§

See FIDO Security Reference document [FIDOSecRef].

Index§

Terms defined by this specification§

- advertisement suffix
- Client Platform
- CMHD
- Credential Manager Hosting Device
- CTAP2 canonical CBOR encoding form
- qr data
- server psm
- tunnel service

Terms defined by reference§

[CTAP] defines the following terms:

- authenticatorGetInfo

commands

[DIGITAL-CREDENTIALS] defines the following terms:

Digital Credentials API

[JSON-SCHEMA] defines the following terms:

JSON Schema

[WEBAUTHN-3] defines the following terms:

PublicKeyCredentialRpEntity

authenticator

RP ID

References§

Normative References§

[BT-EXT-ADV]
Bluetooth Core Specification, Version 5.4: Vol 6, Part B, Link Layer Specification, Section 2.3.4, Common Extended Advertising Payload Format. URL: https://www.bluetooth.com/wp-content/uploads/Files/Specification/HTML/Core-54/out/en/low-energy-controller/link-layer-specification.html#UUID-0c585610-6f9f-3bae-94e0-e3e9254721db

[BT-L2CAP]
Bluetooth Core Specification, Version 5.4: Vol 3, Part A, Logical Link Control and Adaptation Protocol Specification. URL: https://www.bluetooth.com/wp-content/uploads/Files/Specification/HTML/Core-54/out/en/host/logical-link-control-and-adaptation-protocol-specification.html

[CMVP]
Implementation Guidance for FIPS 140-2 and the Cryptographic Module Validation Program - CMVP. December 3, 2019. URL: https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Module-Validation-Program/documents/fips140-2/FIPS1402IG.pdf

[CommonCriteria]
CCRA Members. *Common Criteria Publications*. Work in Progress. URL: http://www.commoncriteriaportal.org/cc/

[CREDENTIAL-MANAGEMENT-1]
Nina Satragno; Marcos Caceres. *Credential Management Level 1*. URL: https://w3c.github.io/webappsec-credential-management/

[CSPN]
CSPN certification, Produits, Formulaires et Méthodologies. URL: https://www.ssi.gouv.fr/administration/produits-certifies/cspn/les-procedures-formulaires-et-methodologies/

[DIGITAL-CREDENTIALS]
Marcos Caceres; Tim Cappalli; Mohamed Amir Yosef. *Digital Credentials*. URL: https://w3c-fedid.github.io/digital-credentials/

[FIDOCTAP]
J. Bradley; et al. *Client to Authenticator Protocol*. 29 May 2026. URL: https://fidoalliance.org/specs/fido-v2.3.1-wd-20260529/fido-client-to-authenticator-protocol-v2.3.1-wd-20260529.html

[FIDOSecRef]
R. Lindemann; et al. *FIDO Security Reference*. 23 May 2022. Proposed Standard. URL: https://fidoalliance.org/specs/common-specs/fido-security-ref-v2.1-ps-20220523.html

[IANA-URI-Schemes]
Uniform Resource Identifier (URI) Schemes. URL: https://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml

[JSON-SCHEMA]
Austin Wright; et al. *JSON Schema: A Media Type for Describing JSON Documents*. 10 June 2022. Internet-Draft. URL: https://datatracker.ietf.org/doc/html/draft-bhutton-json-schema

[RFC7595]

D. Thaler, Ed.; T. Hansen; T. Hardie. *Guidelines and Registration Procedures for URI Schemes*. June 2015. Best Current Practice. URL: https://www.rfc-editor.org/info/rfc7595/

[RFC8949]
C. Bormann; P. Hoffman. *Concise Binary Object Representation (CBOR)*. December 2020. RFC. URL: https://www.rfc-editor.org/rfc/rfc8949.html

[WEBAUTHN-3]
Akshay Kumar; et al. *Web Authentication: An API for accessing Public Key Credentials - Level 3* URL: https://w3c.github.io/webauthn/

Non-Normative References§

[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels* March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[RFC5869]
H. Krawczyk; P. Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. May 2010. Informational. URL: https://www.rfc-editor.org/info/rfc5869/

[RFC8701]
D. Benjamin. *Applying Generate Random Extensions And Sustain Extensibility (GREASE) to TLS Extensibility*. January 2020. Informational. URL: https://www.rfc-editor.org/info/rfc8701/