[MS-CDP]:

Connected Devices Platform Protocol Version 3

Revision Summary

Date	Revision History	Revision Class	Comments
7/14/2016	1.0	Major	Initial Availability

Table of Contents

1 Introduction

The Connected Devices Platform Service Protocol provides a way for devices such as PC's and smartphones to discover and send messages between each other. It provides a transport-agnostic means of building connections among all of a user's devices and allows them to communicate over a secure protocol. There are multiple ways for users to authenticate and when **authentication** is successful, the two devices can communicate over any available transport.

Sections 1.5, 1.8, 1.9, 2, and 3 of this specification are normative. All other sections and examples in this specification are informative.

1.1 Glossary

This document uses the following terms:

Advanced Encryption Standard (AES): A block cipher that supersedes the Data Encryption Standard (DES). AES can be used to protect electronic data. The AES algorithm can be used to encrypt (encipher) and decrypt (decipher) information. **Encryption** converts data to an unintelligible form called ciphertext; decrypting the ciphertext converts the data back into its original form, called plaintext. AES is used in symmetric-key cryptography, meaning that the same **key** is used for the **encryption** and decryption operations. It is also a block cipher, meaning that it operates on fixed-size blocks of plaintext and ciphertext, and requires the size of the plaintext as well as the ciphertext to be an exact multiple of this block size. AES is also known as the Rijndael symmetric encryption algorithm [FIPS197].

authentication: The ability of one entity to determine the identity of another entity.

base64 encoding: A binary-to-text encoding scheme whereby an arbitrary sequence of bytes is converted to a sequence of printable ASCII characters, as described in [RFC4648].

Beacon: A management frame that contains all of the information required to connect to a network. In a WLAN, Beacon frames are periodically transmitted to announce the presence of the network.

big-endian: Multiple-byte values that are byte-ordered with the most significant byte stored in the memory location with the lowest address.

Bluetooth (BT): A wireless technology standard which is managed by the Bluetooth Special Interest Group and that is used for exchanging data over short distances between mobile and fixed devices.

Bluetooth Low Energy (BLE): A low energy version of Bluetooth that was added with Bluetooth 4.0 to enable short burst, short range communication that preserves power but allows proximal devices to communicate.

cipher block chaining (CBC): A method of encrypting multiple blocks of plaintext with a block cipher such that each ciphertext block is dependent on all previously processed plaintext blocks. In the **CBC** mode of operation, the first block of plaintext is XOR'd with an Initialization Vector (IV). Each subsequent block of plaintext is XOR'd with the previously generated ciphertext block before encryption with the underlying block cipher. To prevent certain attacks, the IV must be unpredictable, and no IV should be used more than once with the same key. **CBC** is specified in [SP800-38A] section 6.2.

encryption: In cryptography, the process of obscuring information to make it unreadable without special knowledge.

Hash-based Message Authentication Code (HMAC): A mechanism for message **authentication** using cryptographic hash functions. HMAC can be used with any iterative cryptographic hash function (for example, MD5 and SHA-1) in combination with a secret shared key. The cryptographic strength of HMAC depends on the properties of the underlying hash function.

initialization vector: A data block that some modes of the AES cipher block operation require as an additional initial data input. For more information, see [SP800-38A].

key: In cryptography, a generic term used to refer to cryptographic data that is used to initialize a cryptographic algorithm. **Keys** are also sometimes referred to as keying material.

Microsoft Account: A credential for Windows devices and Microsoft services used to sign in users and connect all of their Microsoft-related products.

private key: One of a pair of keys used in public-key cryptography. The private key is kept secret and is used to decrypt data that has been encrypted with the corresponding public key. For an introduction to this concept, see [CRYPTO] section 1.8 and [IEEE1363] section 3.1.

public key: One of a pair of keys used in public-key cryptography. The public key is distributed freely and published as part of a digital certificate. For an introduction to this concept, see [CRYPTO] section 1.8 and [IEEE1363] section 3.1.

session key: A relatively short-lived symmetric key (a cryptographic key negotiated by the client and the server based on a shared secret). A **session key's** lifespan is bounded by the session to which it is associated. A **session key** has to be strong enough to withstand cryptanalysis for the lifespan of the session.

SHA-256: An algorithm that generates a 256-bit hash value from an arbitrary amount of input data, as described in [FIPS180-2].

SHA-256 hash: The value computed from the hashing function described in [FIPS180-3].

Transmission Control Protocol (TCP): A protocol used with the Internet Protocol (IP) to send data in the form of message units between computers over the Internet. TCP handles keeping track of the individual units of data (called packets) that a message is divided into for efficient routing through the Internet.

Uniform Resource Identifier (URI): A string that identifies a resource. The URI is an addressing mechanism defined in Internet Engineering Task Force (IETF) Uniform Resource Identifier (URI): Generic Syntax [RFC3986].

User Datagram Protocol (UDP): The connectionless protocol within TCP/IP that corresponds to the transport layer in the ISO/OSI reference model.

web service: A service offered by a server to other devices, to allow communication over the web.

MAY, SHOULD, MUST, SHOULD NOT, MUST NOT: These terms (in all caps) are used as defined in [RFC2119]. All statements of optional behavior use either MAY, SHOULD, or SHOULD NOT.

1.2 References

Links to a document in the Microsoft Open Specifications library point to the correct section in the most recently published version of the referenced document. However, because individual documents in the library are not updated at the same time, the section numbers in the documents may not match. You can confirm the correct section numbering by checking the Errata.

1.2.1 Normative References

We conduct frequent surveys of the normative references to assure their continued availability. If you have any issue with finding a normative reference, please contact dochelp@microsoft.com. We will assist you in finding the relevant information.

[MS-DTYP] Microsoft Corporation, "Windows Data Types".

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, http://www.rfc-editor.org/rfc/rfc2119.txt

1.2.2 Informative References

None.

1.3 Overview

With multiple possible transports for Connected Devices Platform V3 service, the protocol defines the discovery system to authenticate and verify users and devices as well as the message exchange between two devices. There will be user-intent to initiate discovery – where a device will listen to broadcasts and authorize device. This device becomes a client in our architecture and the discovered device becomes the host. When a connection is authorized, a transport channel is created between the client and host so that clients can start exchanging messages with the host.

Clients can launch **URIs** and build app services connections between hosts. The following diagram provides an overview of the app communication channels between two devices running the Connected Apps & Devices Platform.

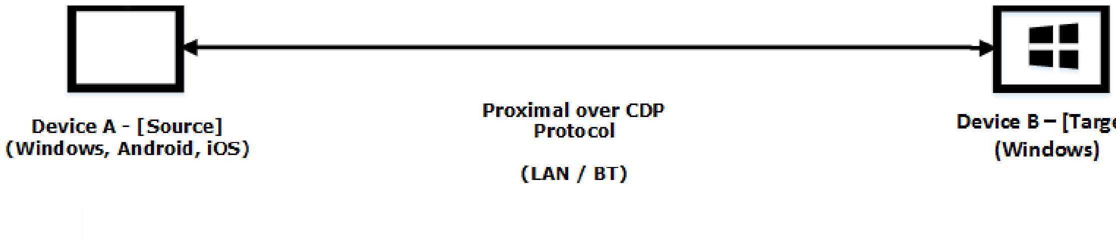

Figure 1: Proximal Communication over CDP Protocol

Launch and Messaging between two devices can occur over proximal connections. Device B (target) acts as the host for the Launch or App Service which can accept incoming client connections from Windows, Android, or iOS devices (source).

1.3.1 Setup

Prior to CDP being used, each device sets up a key-pair to secure communications. A key-pair is the association of a **public key** and its corresponding **private key** when used in cryptography.

1.3.2 Discovery

As described earlier, a client first sends a presence request to the network via broadcast and multicast and starts listening over **Bluetooth Low Energy (BLE)**. This can include parameters and properties to any host that receives the broadcast, which the host can use to evaluate whether to respond. The client then receives unicast responses and can generate the list of available devices. In terms of BLE, devices are constantly advertising a thumbprint that a listener can understand.

1.3.3 Connection

After a device is discovered, the client sends a protocol message to verify that the protocol is supported between both devices. The client derives a **session key** and a public key and sends a connection request. The host receives this request and derives the session key before responding. Finally, the client initiates authorization– the server provides authorization schemes and the client constructs the payload and completes the challenge. The server returns the pairing state and then devices are connected for launch and message exchange.

1.4 Relationship to Other Protocols

None.

1.5 Prerequisites/Preconditions

Peers have to be able to communicate with one of our **web services** in order to obtain information about other devices singed in with the same **Microsoft Account**. In order to fully establish a channel with this protocol, two devices have to be signed-in with the same Microsoft Account. This is a restriction that can be later loosened within the protocol's implementation.

1.6 Applicability Statement

The Connected Devices Platform Service Protocol provides a way for devices such as PCs and smartphones to discover and send messages between each other. It provides a transport-agnostic means of building connections among all of a user's devices, whether available through available transports.

1.7 Versioning and Capability Negotiation

This document is focused on the third version of the protocol (V3)—the protocol version is contained in the header of the messages.

1.8 Vendor-Extensible Fields

None.

1.9 Standards Assignments

None

2 Messages

2.1 Transport

As stated earlier in this document, this protocol can be used for multiple transports. A specific transport is not defined for these messages. Bluetooth Low Energy (BLE), **Bluetooth**, and LAN are all currently supported.

However, the general requirements for a transport are as follows:

- The transport MUST be able to provide the size of each message, independently of its payload, to the component that implements the protocol. Messages are sent and received over the transport on ports that are analogous to ports in **TCP** /IP. Well-known ports allow two peers to establish initial communication.

2.2 Message Syntax

2.2.1 Namespaces

None.

2.2.2 Common Data Types

The data types in the following sections are as specified in [MS-DTYP].

2.2.2.1 Headers

The methods in this protocol use the following headers as part of the information exchanged, prior to any requests or responses that are included in the exchange.

2.2.2.1.1 Common Header

Each channel is responsible for defining its own inner protocol and message types.

Message deserialization is split into two phases. The first phase consists of parsing the header, validating authenticity, deduping, and decryption. The inner buffer is sent to the owner to manage the second part of the deserialization.

<table>
<tr><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>1
0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>2
0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>3
0</th><th>1</th></tr>
<tr><td colspan="16">Signature</td><td colspan="16">Length</td></tr>
<tr><td colspan="8">Version</td><td colspan="8">Message Type</td><td colspan="16">Flags</td></tr>
<tr><td colspan="32">Seq No</td></tr>
<tr><td colspan="32">Request ID</td></tr>
<tr><td colspan="32">...</td></tr>
<tr><td colspan="8">FragmentIndex</td><td colspan="8">FragmentCount</td><td colspan="16">Session</td></tr>
</table>

<table>
<tr><td colspan="4">...</td></tr>
<tr><td colspan="2">...</td><td colspan="2">Channel</td></tr>
<tr><td colspan="4">...</td></tr>
<tr><td colspan="2">...</td><td>Next Header</td><td>Next Header Size</td></tr>
<tr><td colspan="4">Payload (variable)</td></tr>
<tr><td colspan="4">...</td></tr>
<tr><td colspan="4">...</td></tr>
<tr><td colspan="4">HMAC (variable)</td></tr>
<tr><td colspan="4">...</td></tr>
<tr><td colspan="4">...</td></tr>
</table>

Common header fields are described in the following table.

Field	Type	Description
Signature	UINT16	Fixed signature.
Length	UINT16	Entire message length in bytes include signature.
Version	UINT8	Protocol version the sender is speaking.
Message Type	UINT8	Discovery, Connection, or Session message.
Flags	UINT16	ShouldAck | HasHMAC | SessionEncrypted | ChannelEncrypted
Seq No	UINT32	Current message number for this session.
RequestID	UINT64	Request ID – not required.
FragmentIndex	UINT8	Current fragment for current message.
FragmentCount	UINT8	Number of total fragments for current message.
Session	UINT64	ID representing the session.
Channel	UINT64	0 if sessionId == 0
Next Header	enum[UINT8]	If an additional header record is included, this indicates the type. 0 indicates no more header records.
Next Header Size	UINT8	Amount of data in the next header record (so clients can skip).
Payload	array[UINT8]	Encrypted payload.
HMAC	UINT8[32]	Not present if Flags::HasHMAC is not set Only *required* for Control and Session messages.

Each channel is responsible for defining its own inner protocol & message types.

Message deserialization will therefore be split into two phases. With the first phase consisting of the parsing header, validating authenticity, deduping and decryption. The inner buffer will be passed up to the owner to manage the second part of the deserialization.

2.2.2.2 Discovery Messages

For **User Datagram Protocol (UDP)**, a device sends out a presence request and a second device responds with presence response message. For Bluetooth, devices advertise over a **beacon**, which does not require discovery.

2.2.2.2.1 UDP: Presence Request

This is the UDP presence request message – any device can subscribe to and respond to these messages in order to participate in the Connected Devices Protocol message exchange.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>1
0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>2
0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>3
0</td><td>1</td></tr>
<tr><td colspan="8">Message Type</td><td colspan="8">Discovery Type</td></tr>
</table>

Field	Type	Description
Message Type	UINT8	Indicates current message type – in this case, Discovery. This comes from the common header. Values: None = 0, Discovery = 1, Connect = 2, Control = 3, Session = 4, Ack = 5
Discovery Type	UINT8	Indicates type of discovery message, in this case, Presence Request. Presence Request = 0, Presence Response = 1

2.2.2.2.2 UDP: Presence Response

When a device receives a presence request, it responds with a presence response to notify that it's available.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>1
0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>2
0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>3
0</td><td>1</td></tr>
<tr><td colspan="8">Message Type</td><td colspan="8">Discovery Type</td><td colspan="16">Connect Mode</td></tr>
</table>

Device Type	Device Name Length
Device Name (variable)	
...	
...	

Fields are described in the following table.

Field	Type	Description
Message Type	UINT8	Indicates current message type – in this case, Discovery (1).
Discovery Type	UINT8	Indicates type of discovery message, in this case, Presence Response (1).
Connect Mode	UINT16	Displays types of available connections. None = 0, Proximal = 1, Legacy = 2
Device Type	UINT16	SKU of the device (Xbox, Windows): XboxOne = 1, Windows10Desktop = 9, Windows10Phone = 11
Device Name Length	UINT16	Length of the machine name of the device.
Device Name	array[UINT8]	This is character representation of the name of the device. The size of the list is bounded by the previous message.

2.2.2.2.3 Bluetooth: Advertising Beacon

This is the basic beacon structure:

0	1	2	3	4	5	6	7	8	9	1 0	1	2	3	4	5	6	7	8	9	2 0	1	2	3	4	5	6	7	8	9	3 0	1
Len								0xFF								Microsoft ID															
Beacon Data (24 bytes)																															
...																															

<table>
<tr><td colspan="32">...</td></tr>
</table>

The beacon data section is further broken down. Note that the Scenario & Subtype Specific Data section requirements will differ based on the Scenario and Subtype.

<table>
<tr><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>1
0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>2
0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>3
0</th><th>1</th></tr>
<tr><td colspan="8">Scenario Type</td><td colspan="8">Version & Device Type</td><td colspan="8">Version and Flags</td><td colspan="8">Reserved</td></tr>
<tr><td colspan="32">Salt</td></tr>
<tr><td colspan="32">Device Hash</td></tr>
<tr><td colspan="32">...</td></tr>
<tr><td colspan="32">...</td></tr>
<tr><td colspan="32">...</td></tr>
</table>

The descriptions of each field are below:

Field	Type	Description
Length	UINT8	Set to 31
Microsoft ID	UINT8	Set to 0006
Scenario Type	UINT8	Set to 1
Version & Device Type	UINT8	High 2 bits = 00 (version) Lower 6 bits are device type: XboxOne = 1, Windows10Desktop = 9, Windows10Phone = 11
Version & Flags	UINT8	High 3 bits = 001 Lower 3 bits = 00000
Reserved	UINT8	Reserved is currently 0
Salt	array[UINT8]	4 Random Bytes
Hash	array[UINT8]	SHA256 Hash of Salt + Device Thumbprint. Truncated to 16 bytes.

2.2.2.3 Connection Messages

These are the messages during authentication of a connection when a device is discovered.

2.2.2.3.1 Connection Header

The Connection Header is common for all Connection Messages

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
Connect Type								Connect Mode																							

Connection header fields are described in the following table. The message type is used to segment the flow for each type of message.

Field	Type	Description
Connect Type	UINT8	Indicates the current Connection type. ConnectRequest = 0, ConnectResponse = 1, DeviceAuthRequest = 2, DeviceAuthResponse =3, UserDeviceAuthRequest = 4, UserDeviceAuthResponse = 5, AuthDoneRequest = 6, AuthDoneRespone = 7, Failure = 8
Connect Mode	UINT8	Displays types of available connections. None = 0, Proximal = 1, Legacy = 2

Each connection can be the following types:

Connection Type	Description
Connect Request	Device issued connection request (0).
Connect Response	Response to connection request (1)
Device Authentication Request	Initial authentication (Device Level) (2)
Device Authentication Response	Response to initial authentication (3)
User Device Authentication Request	Authentication of user and device combination (depending on authentication model) (4)
User Device Authentication Response	Response to authentication of a user and device combination (depending on authentication model) (5)
Authentication Done Request	Authentication Completed Message (6)
Authentication Done Response	Authentication Completed Response (7)
Authentication Failure	Connection Failed (8)

2.2.2.3.2 Connection Request

Client initiates a connection request with a host device.

<table>
<tr><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>20</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>30</th><th>1</th></tr>
<tr><td colspan="8">Curve Type</td><td colspan="16">HMACSize</td><td colspan="8">Nonce</td></tr>
<tr><td colspan="32">Nonce</td></tr>
<tr><td colspan="24">Nonce</td><td colspan="8">Message Fragment Size</td></tr>
<tr><td colspan="24">Message Fragment Size</td><td colspan="8">Public Key X Length</td></tr>
<tr><td colspan="8">Public Key X Length</td><td colspan="24">Public Key X</td></tr>
<tr><td colspan="32">...</td></tr>
<tr><td colspan="32">...</td></tr>
<tr><td colspan="16">Public Key Y Length</td><td colspan="16">Public Key Y (variable)</td></tr>
<tr><td colspan="32">...</td></tr>
<tr><td colspan="32">...</td></tr>
</table>

Field	Type	Description
Curve Type	UINT8	Type of elliptical curve used. CT_NIST_P256_KDF_SHA512 = 0
HMAC Size	UINT32	Expected size of HMAC (see more in Encryption)
Nonce	UINT64	Random Values (see more in Encryption)
Message Fragment Size	UINT32	Maximum size of a single message fragment (Fixed Value of 16384)
Public Key X Component Length	UINT16	Length of Public Key X.
Public Key X Component	array[UINT8]	Fixed length key based on Public Key X Length.
Public Key Y Component Length	UINT16	Length of Public Key Y.
Public Key Y Component	array[UINT8]	Fixed length key based on Public Key Y Length.

2.2.2.3.3 Connection Response

The host responds with a connection response message including device information.

Only the Result is sent if the Result is anything other than PENDING.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>1
0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>2
0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>3
0</td><td>1</td></tr>
<tr><td colspan="8">Result</td><td colspan="16">HMACSize</td><td colspan="8">Nonce</td></tr>
<tr><td colspan="32">Nonce</td></tr>
<tr><td colspan="24">Nonce</td><td colspan="8">Message Fragment Size</td></tr>
<tr><td colspan="24">Message Fragment Size</td><td colspan="8">Public Key X Length</td></tr>
<tr><td colspan="8">Public Key X Length</td><td colspan="24">Public Key X</td></tr>
<tr><td colspan="32">...</td></tr>
<tr><td colspan="32">...</td></tr>
<tr><td colspan="16">Public Key Y Length</td><td colspan="16">Public Key Y (variable)</td></tr>
<tr><td colspan="32">...</td></tr>
<tr><td colspan="32">...</td></tr>
</table>

Field	Type	Description
Result	UINT8	Success, Failure, or Other of the connection request. Success = 0, Pending = 1, Failure_Authentication = 2, Failure_NotAllowed
HMAC Size	UINT32	Expected size of HMAC (see more in Encryption)
Nonce	UINT64	Random Values (see more in Encryption)
Message Fragment Size	UINT32	Maximum size of a single message fragment (Fixed Value of 16384)
Public Key X Component Length	UINT16	Length of Public Key X Component– sent only if connection is successful.
Public Key X Component	array[UINT8]	Fixed length key based on curve type from connect request – sent only if connection is successful. This is the X component of the Key.
Public Key Y Component Length	UINT16	Length of Public Key Component Y – sent only if connection is

Field	Type	Description
		successful.
Public Key Y Component	array[UINT8]	Fixed length key based on curve type from connect request – sent only if connection is successful. This is the Y component of the Key.

2.2.2.3.4 Device Authentication Request

For all authentication, client devices send their device certificate, which is self-signed.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>20</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>30</td><td>1</td></tr>
<tr><td colspan="16">Cert Length</td><td colspan="16">Cert (variable)</td></tr>
<tr><td colspan="32">...</td></tr>
<tr><td colspan="32">...</td></tr>
<tr><td colspan="16">Thumbprint Length</td><td colspan="16">Thumbprint (variable)</td></tr>
<tr><td colspan="32">...</td></tr>
<tr><td colspan="32">...</td></tr>
</table>

Field	Type	Description
Cert Length	UINT16	Length of Device Certificate.
Cert	array[UINT8]	Device Certificate.
Thumbprint Length	UINT16	Length of Device Cert Thumbprint.
Thumbprint	array[UINT8]	Signed Device Cert Thumbprint.

2.2.2.3.5 Device Authentication Response

For all authentication, hosts send their device certificate, which is self-signed.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>20</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>30</td><td>1</td></tr>
<tr><td colspan="16">Cert Length</td><td colspan="16">Cert (variable)</td></tr>
<tr><td colspan="32">...</td></tr>
</table>

...

Thumbprint Length	Thumbprint (variable)

...
...

Field	Type	Description
Cert Length	UINT16	Length of Device Certificate.
Cert	array[UINT8]	Device Certificate.
Thumbprint Length	UINT16	Length of Device Cert Thumbprint.
Thumbprint	array[UINT8]	Signed Device Cert Thumbprint.

2.2.2.3.6 User-Device Authentication Request

If authentication policy requires user-device authentication, the user-device certificate is sent with the request.

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
Cert Length																Cert (variable)															
...																															
...																															
Thumbprint Length																Thumbprint (variable)															
...																															
...																															

Field	Type	Description
Cert Length	UINT16	Length of User-Device Certificate.
Cert	array[UINT8]	User-Device Certificate.
Thumbprint Length	UINT16	Length of User-Device Cert Thumbprint.

Field	Type	Description
Thumbprint	array[UINT8]	Signed User-Device Cert Thumbprint.

2.2.2.3.7 User-Device Authentication Response

If authentication policy requires user-device authentication, the user-device certificate is sent with the request.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>20</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>30</td><td>1</td></tr>
<tr><td colspan="16">Cert Length</td><td colspan="16">Cert (variable)</td></tr>
<tr><td colspan="32">...</td></tr>
<tr><td colspan="32">...</td></tr>
<tr><td colspan="16">Thumbprint Length</td><td colspan="16">Thumbprint (variable)</td></tr>
<tr><td colspan="32">...</td></tr>
<tr><td colspan="32">...</td></tr>
</table>

Field	Type	Description
Cert Length	UINT16	Length of User-Device Certificate.
Cert	array[UINT8]	User-Device Certificate
Thumbprint Length	UINT16	Length of User-Device Cert Thumbprint
Thumbprint	array[UINT8]	Signed User-Device Cert Thumbprint

2.2.2.3.8 Authentication Done Request

Message to acknowledge that Authentication was completed.

Empty Payload.

2.2.2.3.9 Authentication Done Response

Message to respond with the status of authentication.

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
Status																															

Field	Type	Description
Status	UINT8	Status of Authentication Success = 0, Pending = 1, Failure_Authentication = 2, Failure_NotAllowed = 3, Failure_Unknown = 4

2.2.2.3.10 Authentication Failure

Empty Payload.

2.2.2.4 Session Messages

These messages are sent across during an active session between two connected and authenticated devices.

2.2.2.4.1 Ack Messages

These messages acknowledge receipt of a message.

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
Low Watermark																															
Processed (variable)																															
...																															
...																															
Rejected (variable)																															
...																															
...																															

Field	Type	Description
Low Watermark	UINT32	The sequence number of the latest acknowledged message.
Processed	array[UINT32]	The sequence numbers of messages that have been processed.
Rejected	array[UINT32]	The sequence numbers of messages that were rejected.

2.2.2.4.2 App Control Messages

There are four types of app control messages that are used: LaunchURI, LauncURIResponse, CallAppService, and CallAppService Response.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>20</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>30</td><td>1</td></tr>
<tr><td colspan="8">Message Type</td></tr>
</table>

Field	Type	Description
App Control Message Type	UINT8	The type of app control message. LaunchURI - 0 LaunchURIResult - 1 CallAppService - 6 CallAppServiceResponse – 7

2.2.2.4.2.1 Launch URI Messages

These messages allow you to launch apps on CDP-enabled devices—this simply launches using the LaunchURIAsync API in Windows.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>20</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>30</td><td>1</td></tr>
<tr><td colspan="16">URI Length</td><td colspan="16">URI (variable)</td></tr>
<tr><td colspan="32">...</td></tr>
<tr><td colspan="32">...</td></tr>
<tr><td colspan="16">Launch Location</td><td colspan="16">Request ID</td></tr>
<tr><td colspan="32">Request ID</td></tr>
</table>

<table>
<tr><td colspan="16">Request ID</td><td colspan="16">Input Data Length</td></tr>
<tr><td colspan="16">Input Data Length</td><td colspan="16">Input Data…(variable)</td></tr>
<tr><td colspan="32">...</td></tr>
<tr><td colspan="32">...</td></tr>
</table>

2.2.2.4.2.2 Launch URI Result

This returns the result of the LaunchUriAsync API call on the second device.

<table>
<tr><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>1
0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>2
0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>3
0</th><th>1</th></tr>
<tr><td colspan="32">Launch URI Result</td></tr>
<tr><td colspan="32">Response ID</td></tr>
<tr><td colspan="32">Response ID</td></tr>
<tr><td colspan="32">Input Data…(variable)</td></tr>
<tr><td colspan="32">...</td></tr>
<tr><td colspan="32">...</td></tr>
</table>

2.2.2.4.2.3 App Service Message

These messages allow background invocation of background services within apps.

<table>
<tr><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>1
0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>2
0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>3
0</th><th>1</th></tr>
<tr><td colspan="16">Package Name Length</td><td colspan="16">Package Name (variable)</td></tr>
<tr><td colspan="32">...</td></tr>
<tr><td colspan="32">...</td></tr>
<tr><td colspan="16">App Service Name Length</td><td colspan="16">App Service Name (variable)</td></tr>
<tr><td colspan="32">...</td></tr>
<tr><td colspan="32">...</td></tr>
</table>

Return Data (variable)
...
...

Field	Type	Description
Package Name Length	UINT16	Length of Package Name
Package Name	array[UINT8]	Package Name of the app that hosts the app service (Chars)
AppServiceName Length	UINT16	Length of App Service Name
App Service Name	array[UINT8]	Name of App Service (Chars)
Input Data	array[UINT8]	List of parameters sent to App Service for execution

2.2.2.4.2.4 App Services Result

This returns the result of the App Services API call from the second device.

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
App Services Result Result URI																															
Return Data…Input Data (variable)																															
...																															
...																															

2.3 Directory Service Schema Elements

None.

3 Protocol Details

3.1 Peer Details

This section defines peer roles in the Connected Devices Platform V3 Service Protocol.

In a socket-based connection between two peer applications, one peer has the role of client, and the other peer has the role of host. The roles are distinguished as follows:

- The device that performs discovery (and initiates connection) is the client. For UDP, this device sends the **Presence Request** message as well as the **Connection Request** message. For BLE, this device scans for beacons.
- The host is the peer that is discovered (and is the connection target). For UDP, this device receives the **Presence Request** message and sends back a **Presence Response** message. It also receives the **Connection Request** message and responds. For BLE, this is the device that advertises its beacon.

During a connection, these two devices communicate by sending messages back and forth and requesting/requiring **Ack** messages when necessary. All messages during a connection are contained in **Session Messages**.

3.1.1 Abstract Data Model

This section describes a conceptual model of possible data organization that an implementation maintains to participate in this protocol. The described organization is provided to facilitate the explanation of how the protocol behaves. This document does not mandate that implementations adhere to this model as long as their external behavior is consistent with that described in this document.

The abstract data model defines the **peers**, **client** and **host**, as well as the **session** (connections between a **client** and **host**), and **connections**. When one device discovers another, the device can trigger a **connection**. If the connection is successful, based on authentication, each peer creates a **session**. At this point, the objects act more as **peers** than **clients** and **hosts**.

3.1.1.1 CDP Service

The Connected Devices Platform service, **CDPService**, contains the entire state of the protocol described in this object.

3.1.1.2 Discovery Object

The **Discovery** object encapsulates the state for the discovery of one peer from another. Again, the discovering peer is the client and the discovered peer is the host.

Roles: One peer is the client and the other peer is the host.

- The client is the peer that sends the **Presence Reques**t message and waits for the **Presence Response Message**.
- The host is the peer that receives the **Presence Request** message and sends the **Presence Response Message**.

Client State: The current role of the **Discovery** object. For the client, the state can be one of the following values:

Value	Meaning
Waiting for Presence Response	The object has published the **Presence Request** message (section 2.2.2.2.1) and is waiting to receive the **Presence Response** message (section 2.2.2.2.2).
Ready	The object has received the **Presence Response** message and has the basic information it needs to request a connection with the other peer.

Host State: The current role of the **Discovery** object. For the host, the state can be one of the following values:

Value	Meaning
Waiting for Presence Request	The object is waiting to receive the **Presence Response** message (section 2.2.2.2.2).
Ready	The object has sent the **Presence Response** message and has sent the basic information it to facilitate a connection request.

3.1.1.3 Connection Manager Object

The **Connection Manager** object encapsulates the state for the connection between one peer and another. Again, the connecting peer is the client and the peer hosting the connection is the host.

Roles: One peer is the client and the other peer is the host.

- The client is the peer that sends the **Connection Request** message and waits for the **Connection Response Message**.
- The host is the peer that receives the **Connection Request** message and sends the **Connection Response Message**.

Client State: The current role of the **Connection Manager** object. For the client, the state can be one of the following values:

Value	Meaning
Waiting for Connection Response	The object has published the **Connection Request** message (section 2.2.2.3.2) and is waiting to receive the **Connection Response** message (section 2.2.2.3.3).
Connection Failed	The connection has failed – either the **Connection Request** message (section 2.2.2.3.2) has timed out or **Authentication** has failed.
Waiting for Authentication Response	The object has received the **Connection Response** message (section 2.2.2.3.3) and has published the **Authentication Request** message
Ready	The object has received the **Authentication Response** message and is ready to initiate the session with the peer.

Host State: The current role of the **Connection Manager** object. For the host, the state can be one of the following values:

Value	Meaning
Waiting for Connection Request	The object has published the **Presence Response** message (section 2.2.2.2.2) and is waiting to receive the **Connection Request** message (section 2.2.2.3.2).
Waiting for Authentication Request	The object has received the **Connection Request** message and has published the **Connection Response** message – which contains an **Authentication Challenge.** It's waiting for an **Authentication Request**.
Connection Failed	The object has received the **Authentication Request** and the connecting device has failed authentication.
Ready	The object has published the **Authentication Response** message and is ready to engage in a session with the peer.

3.1.1.4 Session Object

A **Session** object encapsulates the state for a socket-based connection between two peer applications.

Role: The role of the **Session** object. Both peers essentially play the same role since either can initiate or receive a message.

State: The current state of the **Session** object. The state can be one of the following.

Value	Meaning
WaitingForAck	A **Session** object transitions to this state immediately prior to publishing a **Session** message. This is not always required for each type of message.
WaitingForTransmit	A **Session** object transitions to this state when beginning to publish the **Session ACK** message.
Ready	The **Session** object is ready to be used by an application for peer-to-peer communication. A client **Session** object transitions to this state after receiving the **Session ACK** message. A server **Session** object transitions to this state after successfully transmitting the **Session ACK** message.
Terminated	The **Session** object has been terminated by the application, or it timed out.

3.1.2 Timers

Heartbeat timer: The heartbeat timer is used to track whether a **session** is still alive. If two peers are not actively sending or receiving messages, heartbeat timers verify the connection between the two peers is still alive.

Message Timer: A timeout indicating that we have not received the requested ACK for a particular message. While sending a message, an ACK can be requested – if it is, the service starts a timer to verify that a response is received in time.

3.1.3 Initialization

The **CDPService** MUST be initialized prior to being useful for any discovery, connection, or sessions; initializing at system startup and signing in with a user account is sufficient. On initialization:

- Generation of Device Certificate (on system boot) – this certificate is used as part of authentication between two devices.
- Generation of User-Device Certificate (on system sign-in) – this certificate is used as part of authentication between two devices with the same user.

3.1.3.1 Encryption

During connection establishment, the first connect message from each side is used to trade, amongst other things, random 64-bit nonces. The initiator of the connection is referred to as the client, and his nonce is referred to as the clientNonce. The target of the connection is referred to as the host, and his nonce is referred to as the hostNonce.

The signed thumbprint (from the certificates setup during initialization) that is sent is a **SHA-256 hash** of (hostNonce | clientNonce | cert), where | is the append operator.

Also after the first connection messages are exchanged, an ephemeral Diffie-Hellman secret is created. This secret is then passed into a standard HKDF algorithm to obtain a cryptographically random buffer of 64 bytes. The first 16 bytes are used to create an **encryption key**, the next 16 bytes are used to create an **initialization vector** (IV) key (both are **Advanced Encryption Standard (AES)** 128-bit in **cipher block chaining (CBC)** mode), and the final 32 bytes are used to create a hash (**SHA-256**) with a shared secret that is meant to be used for message authentication (**Hash-based Message Authentication Code (HMAC)**). All messages after the initial connection message exchange are encrypted and verified using a combination of these objects.

The examples in section 4 are unencrypted payloads. Described here is the transformation each message goes through to becoming encrypted.

The payload of each message is considered to be the content beyond the "EndAdditionalHeaders" marker. The payload is prepended with the total size of the payload as an unsigned 4-byte integer. This modified payload's length is then rounded up to a multiple of the encryption algorithm's block length (16 bytes) and is referred to as the to-be-encrypted payload length. The difference between the to-be-encrypted payload length and the modified payload length is referred to as the padding length. The modified payload is then padded to the to-be-encrypted payload length by appending the padding length repeatedly in the remaining space.

The initialization vector for a message is created by encrypting with the IV key the 16-byte payload of the message's session ID, sequence number, fragment number, and fragment count, each in **big-endian** format. This initialization vector is then used with the encryption key as the two parts of the AES-128 CBC algorithm to encrypt the aforementioned to-be-encrypted payload. This payload is the encrypted payload and is of the same length as the to-be-encrypted payload. Once this is completed, the message flag field is binary **OR'd** with the hexadecimal number 0x4 to indicate that it contains an encrypted payload.

The unencrypted header and the entire encrypted message is then hashed with the HMAC algorithm and appended onto the final message. The message flag field is binary **OR'd** with the hexadecimal number 0x2 to indicate that it has a HMAC and should be verified.

The message size field is then set to the sum of the length of the message header (everything before the payload), the encrypted payload length, and the hash length.

3.1.3.1.1 Encryption Example

The following is an example of the process to convert an unencrypted message to an encrypted message.

Unencrypted Message

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>1
0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>2
0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>3
0</td><td>1</td></tr>
<tr><td colspan="16">Signature = 0x30, 0x30</td><td colspan="16">MessageLength = 45 bytes
0x00, 0x2D</td></tr>
<tr><td colspan="8">Version = 0x03</td><td colspan="8">MessageType = Connect
0x02</td><td colspan="16">MessageFlags = None
0x00, 0x00</td></tr>
<tr><td colspan="32">SequenceNumber = 0
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="32">RequestID = 0
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="16">FragmentNumber = 0
0x00, 0x00</td><td colspan="16">FragmentCount = 1
0x00, 0x01</td></tr>
<tr><td colspan="32">SessionID=
0x00, 0x00, 0x00, 0x01
0x00, 0x00, 0x00, 0x01</td></tr>
<tr><td colspan="32">ChannelID = 0
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="16">EndAdditionalHeaders = 0x00, 0x00</td><td colspan="16">ConnectionMode = ECDHE
0x00, 0x01</td></tr>
<tr><td colspan="8">ECDHE::MessageType = AuthenticationDoneRequest
0x06</td></tr>
</table>

Encrypt, using AES 128-bit algorithm in CBC mode with the IV key as described above, the concatenated values of the SessionId, SequenceNumber, FragmentNumber, and FragmentCount.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>20</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>30</td><td>1</td></tr>
<tr><td colspan="32">SessionID =
0x00, 0x00, 0x00, 0x01
0x00, 0x00, 0x00, 0x01</td></tr>
<tr><td colspan="32">SequenceNumber = 0
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="16">FragmentNumber = 0
0x00, 0x00</td><td colspan="16">FragmentCount = 1
0x00, 0x01</td></tr>
</table>

The output of this encryption will be referred to as the initialization vector.

Before encrypting the message payload, the unencrypted payload size is prepended to the payload, and then padded to a length that is a multiple of AES 128-bit CBC's block size (16 bytes). The padding is appended to the new payload and padding value is the difference between the intermediate payload size and the final payload size. Changes from the previous message are marked with **bold**.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>20</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>30</td><td>1</td></tr>
<tr><td colspan="16">Signature = 0x30, 0x30</td><td colspan="16">MessageLength = 58 bytes
0x00, 0x3A</td></tr>
<tr><td colspan="8">Version = 0x03</td><td colspan="8">MessageType = Connect
0x02</td><td colspan="16">MessageFlags = None
0x00, 0x00</td></tr>
<tr><td colspan="32">SequenceNumber = 0
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="32">RequestID = 0
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="16">FragmentNumber = 0
0x00, 0x00</td><td colspan="16">FragmentCount = 1
0x00, 0x01</td></tr>
</table>

SessionID = 0x00, 0x00, 0x00, 0x01 0x00, 0x00, 0x00, 0x01			
ChannelID = 0 0x00, 0x00, 0x00, 0x00 0x00, 0x00, 0x00, 0x00			
EndAdditionalHeaders = 0x00, 0x00			
PayloadSize = **0x00, 0x00, 0x00, 0x03**			
ConnectionMode = ECDHE 0x00, 0x01		**ECDHE::MessageType = AuthenticationDoneRequest**	**Padding = 7 0x07**
Padding = 7 0x07	**Padding = 7 0x07**	**Padding = 7 0x07**	**Padding = 7 0x07**
Padding = 7 0x07	**Padding = 7 0x07**		

This new payload is then encrypted using AES 128-bit CBC using the encryption key and the aforementioned initialization vector (an input of the algorithm). The changes are in **bold**.

Encrypted Message

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
Signature = 0x30, 0x30																MessageLength = 58 bytes 0x00, 0x3A															
Version = 0x03								MessageType = Connect 0x02								MessageFlags = None 0x00, 0x00															
SequenceNumber = 0 0x00, 0x00, 0x00, 0x00																															

RequestID = 0 0x00, 0x00, 0x00, 0x00 0x00, 0x00, 0x00, 0x00			
FragmentNumber = 0 0x00, 0x00		FragmentCount = 1 0x00, 0x01	
SessionID = 0x00, 0x00, 0x00, 0x01 0x00, 0x00, 0x00, 0x01			
ChannelID = 0 0x00, 0x00, 0x00, 0x00 0x00, 0x00, 0x00, 0x00			
EndAdditionalHeaders = 0x00, 0x00			
Encrypted	**Encrypted**	**Encrypted**	**Encrypted**
Encrypted	**Encrypted**	**Encrypted**	**Encrypted**
Encrypted	**Encrypted**	**Encrypted**	**Encrypted**
Encrypted	**Encrypted**		

Finally, the entire message is hashed with a SHA-256 HMAC algorithm, where the secret key comes from the aforementioned secret exchange. This hash is then appended to the message and the message size is updated accordingly. The changes are in **bold**.

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
Signature = 0x30, 0x30																**MessageLength = 90 bytes** **0x00, 0x5A**															

<table>
<tr><td>Version = 0x03</td><td>MessageType = Connect
0x02</td><td colspan="2">MessageFlags = None
0x00, 0x00</td></tr>
<tr><td colspan="4">SequenceNumber = 0
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="4">RequestID = 0
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="2">FragmentNumber = 0
0x00, 0x00</td><td colspan="2">FragmentCount = 1
0x00, 0x01</td></tr>
<tr><td colspan="4">SessionID =
0x00, 0x00, 0x00, 0x01
0x00, 0x00, 0x00, 0x01</td></tr>
<tr><td colspan="4">ChannelID = 0
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="2">EndAdditionalHeaders = 0x00, 0x00</td><td colspan="2"></td></tr>
<tr><td>Encrypted</td><td>Encrypted</td><td>Encrypted</td><td>Encrypted</td></tr>
<tr><td>Encrypted</td><td>Encrypted</td><td>Encrypted</td><td>Encrypted</td></tr>
<tr><td>Encrypted</td><td>Encrypted</td><td>Encrypted</td><td>Encrypted</td></tr>
<tr><td>Encrypted</td><td>Encrypted</td><td></td><td></td></tr>
</table>

SHA 256 Hash (32 bytes)

3.1.4 Higher-Layer Triggered Events

When **CDPService** is inactive for a specific duration (defined by the idle timer), it automatically shuts down to save the system resources. The service wakes up again when there's traffic detected on a specific port or when it's activated through some other means.

3.1.5 Message Processing Events and Sequencing Rules

When a message is received, the type of message is handled and disambiguated at the first level – the three primary message types are Discovery, Connect, and Session respectively. Session messages have to be preceded by Discovery and/or Connect message. If the device is already known (by IP or other means), a discovery message may not be necessary. Message processing is different from the client and host. Each message is verified to make sure the message is of valid format and used sequence numbers are thrown away to prevent handling the same messages twice.

3.1.5.1 Discovery

If the message is a discovery message, the service will do the following, depending on if it is client and host. A client initiates this segment by sending a PresenceRequest message.

Client

1. Send a PresenceRequest to the original device.

Host

1. Verifies the message is a CDP message of type PresenceRequest.
2. Send a PresenceResponse back to the original device.

3.1.5.2 Connection

If the message is a discovery message, the service will do the following, depending on if it is client and host. A client initiates this segment by sending a ConnectionRequest message. The client either needs to discover or already know the endpoint that it is attempting to start a connection with.

Host

1. Verify the message is a Connection message.
2. Determine Session ID for the connection.
3. Determine type of connection (legacy).
4. Determine type of connection message. These must flow in order from ConnectionRequest -> DeviceAuthenticationRequest -> UserDeviceAuthenticationRequest (if necessary) -> AuthenticationDoneRequest. The host will send back appropriate Response messages for each type of message. If anything fails, the connection is dropped.
5. Establish a session when Authentication completes successfully with the given Session ID.

Client

1. Verify the message is a Connection Response message.

2. Read Response results to verify the Response has a successful status and then send the next Request message. This again flows in the order above: ConnectionRequest -> DeviceAuthenticationRequest -> UserDeviceAuthenticationRequest (if necessary) -> AuthenticationDoneRequest.

3.1.5.3 Session

Host

1. Retrieve session ID and verify the session ID has a matching session.

2. Reset heartbeat timer as a result of receiving a message, which verifies the connection still exists.

3. The message is processed and the corresponding API is called (LaunchUriAsync, AppServices, etc.). At this point, a host implementation can take any action on the host device as a result of the message.

Client

1. Wait for messages responses from Host device and optionally request Ack's to determine whether message gets acknowledged.

2. Reset heartbeat timer as a result of receiving a message, which verifies the connection still exists.

3.1.6 Timer Events

The following timer events are associated with the timers defined by this protocol (section 3.1.2).

Heartbeat timer: The heartbeat timer is used to track whether a **session** is still alive. If the heartbeat timer fires during a session, the session is ended.

Message Timer: A timeout indicating that we have not received the requested ACK for a particular message. If this timer fires, the message is resent.

3.1.7 Other Local Events

None.

4 Protocol Examples

The following scenario shows a successful connection established between two peers, Peer A and Peer B.

In the following examples, the hostname of Peer A is "devicers1 -2" and the hostname of Peer B is "devicers1 -1".

Peer A has a 32-byte device ID that has a **base64 encoding** representation of "D3kXI3RR9kYneA2AQuqEgjmeJ21uyCvAAJ5kNjyJx+c=".

Peer B has a 32-byte device ID that has a base64 encoding representation of "l6+4vOa41cFV+CvBEbJtoY5xRfqDoo63l90QGa+HAUw=".

4.1 Discovery

4.1.1 Discovery Presence Request

When discovery on Peer A is activated, it sends the following message, a **Discovery Presence Request**, on all available transports. On IP networks, it chooses to send to the well-defined port 5050. Length = 43 bytes.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>1
0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>2
0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>3
0</td><td>1</td></tr>
<tr><td colspan="16">Signature = 0x30, 0x30</td><td colspan="16">MessageLength = 43 bytes
0x00, 0x2B</td></tr>
<tr><td colspan="8">Version = 0x03</td><td colspan="8">MessageType = Discovery
0x01</td><td colspan="16">MessageFlags = None
0x00, 0x00</td></tr>
<tr><td colspan="32">SequenceNumber = 0
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="32">RequestID = 0
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="16">FragmentNumber = 0
0x00, 0x00</td><td colspan="16">FragmentCount = 1
0x00, 0x01</td></tr>
<tr><td colspan="32">SessionID =
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
</table>

<table>
<tr><td colspan="32">ChannelID = 0
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="16">EndAdditionalHeaders = 0x00, 0x00</td><td colspan="8">DiscoveryMessage::Type = PresenceRequest
0x00</td></tr>
</table>

4.1.2 Discovery Presence Response

When Peer B receives the Discovery Presence Request from Peer A, it proceeds to respond with a **Discovery Presence Response**. On IP networks, this is sent from the well-defined port 5050. Length = 97 bytes.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>20</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>30</td><td>1</td></tr>
<tr><td colspan="16">Signature = 0x30, 0x30</td><td colspan="16">MessageLength = 97 bytes
0x00, 0x61</td></tr>
<tr><td colspan="8">Version = 0x03</td><td colspan="8">MessageType = Discovery
0x01</td><td colspan="16">MessageFlags = None
0x00, 0x00</td></tr>
<tr><td colspan="32">SequenceNumber = 0
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="32">RequestID = 0
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="16">FragmentNumber = 0
0x00, 0x00</td><td colspan="16">FragmentCount = 1
0x00, 0x01</td></tr>
<tr><td colspan="32">SessionID =
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="32">ChannelID = 0
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
</table>

<table>
<tr><td colspan="2">EndAdditionalHeaders = 0x00, 0x00</td><td>DiscoveryMessage::Type = PresenceResponse

0x01</td><td></td></tr>
<tr><td colspan="2">ConnectionMode = ECDHE

0x00, 0x01</td><td colspan="2">DeviceType = Windows10Desktop

0x00, 0x09</td></tr>
<tr><td colspan="2">DeviceNameSize = 11 bytes

0x00, 0x0B</td><td colspan="2"></td></tr>
<tr><td colspan="4">DeviceName = "devicers1-1" (null-terminated)

0x64, 0x65, 0x76, 0x69

0x63, 0x65, 0x72, 0x73</td></tr>
<tr><td colspan="4">...</td></tr>
<tr><td colspan="4">DeviceIdSalt = 0xD6, 0xE7, 0x60, 0x2D</td></tr>
<tr><td colspan="4">DeviceId = SHA256 hash of device id, salted, 32-bytes

0x11, 0x16, 0x6D, 0x8B,

0x4C, 0x02, 0x7A, 0x54</td></tr>
</table>

4.2 Connection

4.2.1 Connection Request

Length = 128 bytes.

<table>
<tr><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>20</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>30</th><th>1</th></tr>
<tr><td colspan="16">Signature = 0x30, 0x30</td><td colspan="16">MessageLength = 128 bytes

0x00, 0x80</td></tr>
<tr><td colspan="8">Version = 0x03</td><td colspan="8">MessageType = Connect

0x02</td><td colspan="16">MessageFlags = None

0x00, 0x00</td></tr>
<tr><td colspan="32">SequenceNumber = 0

0x00, 0x00, 0x00, 0x00</td></tr>
</table>

<table>
<tr><td colspan="4">RequestID = 0
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="2">FragmentNumber = 0
0x00, 0x00</td><td colspan="2">FragmentCount = 1
0x00, 0x01</td></tr>
<tr><td colspan="4">SessionID =
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x01</td></tr>
<tr><td colspan="4">ChannelID = 0
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="2">EndAdditionalHeaders = 0x00, 0x00</td><td colspan="2">ConnectionMode = ECDHE
0x00, 0x01</td></tr>
<tr><td>ECDHE::MessageType = ConnectRequest
0x00</td><td>ECDHE::CurveType = CT NIST P256 KDF SHA512
0x00</td><td colspan="2">HmacSize = 32
0x00, 0x20</td></tr>
<tr><td colspan="4">Nonce =
0x99, 0x1A, 0xF3, 0xCC,
0x7D, 0xE3, 0x41, 0x82</td></tr>
<tr><td colspan="4">MessageFragmentSize = 16384
0x00, 0x00, 0x40, 0x00</td></tr>
<tr><td colspan="2">PublicKeyXBytesLength = 32
0x00, 0x20</td><td colspan="2"></td></tr>
<tr><td colspan="4">PublicKeyXBytes =
0x83, 0xB5, 0x2D, 0xA8,
0xF5, 0x06, 0xD3, 0x01</td></tr>
<tr><td colspan="4">...</td></tr>
</table>

<table>
<tr><td colspan="16">PublicKeyYBytesLength = 32
0x00, 0x20</td><td colspan="16"></td></tr>
<tr><td colspan="32">PublicKeyYBytes =
0xA5, 0x63, 0xF5, 0x10,
0x30, 0xE1, 0x5E, 0xB9</td></tr>
<tr><td colspan="32">...</td></tr>
</table>

4.2.2 Connection Response

Length = 114 bytes.

<table>
<tr><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>1
0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>2
0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>3
0</th><th>1</th></tr>
<tr><td colspan="16">Signature = 0x30, 0x30</td><td colspan="16">MessageLength = 114 bytes
0x00, 0x80</td></tr>
<tr><td colspan="8">Version = 0x03</td><td colspan="8">MessageType = Connect
0x02</td><td colspan="16">MessageFlags = None
0x00, 0x00</td></tr>
<tr><td colspan="32">SequenceNumber = 0
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="32">RequestID = 0
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="16">FragmentNumber = 0
0x00, 0x00</td><td colspan="16">FragmentCount = 1
0x00, 0x01</td></tr>
<tr><td colspan="32">SessionID =
0x00, 0x00, 0x00, 0x01
0x80, 0x00, 0x00, 0x01</td></tr>
</table>

<table>
<tr><td colspan="4">ChannelID = 0
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="2">EndAdditionalHeaders = 0x00, 0x00</td><td colspan="2">ConnectionMode = ECDHE
0x00, 0x01</td></tr>
<tr><td>ECDHE::MessageType = ConnectResponse
0x01</td><td>ConnectRequestResult= Pending
0x01</td><td colspan="2">HmacSize = 32
0x00, 0x20</td></tr>
<tr><td colspan="4">Nonce =
0x18, 0x8A, 0xCB, 0xE0,
0x9F, 0x20, 0x3B, 0x71</td></tr>
<tr><td colspan="4">MessageFragmentSize = 16384
0x00, 0x00, 0x40, 0x00</td></tr>
<tr><td colspan="2">PublicKeyXBytesLength = 32
0x00, 0x20</td><td colspan="2"></td></tr>
<tr><td colspan="4">PublicKeyXBytes =
0x66, 0xD5, 0x2E, 0x11,
0x99, 0xB2, 0xA4, 0x91</td></tr>
<tr><td colspan="4">...</td></tr>
<tr><td colspan="2">PublicKeyYBytesLength = 32
0x00, 0x20</td><td colspan="2"></td></tr>
<tr><td colspan="4">PublicKeyYBytes =
0xB4, 0x13, 0xFA, 0xAA,
0x67, 0x1E, 0xE5, 0x92</td></tr>
<tr><td colspan="4">...</td></tr>
</table>

4.2.3 Device Authentication Request

Length = 500 bytes.

<table>
<tr><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>1
0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>2
0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>3
0</th><th>1</th></tr>
<tr><td colspan="16">Signature = 0x30, 0x30</td><td colspan="16">MessageLength = 500 bytes
0x01, 0xF4</td></tr>
<tr><td colspan="8">Version = 0x03</td><td colspan="8">MessageType = Connect
0x02</td><td colspan="16">MessageFlags = None
0x00, 0x00</td></tr>
<tr><td colspan="32">SequenceNumber = 0
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="32">RequestID = 0
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="16">FragmentNumber = 0
0x00, 0x00</td><td colspan="16">FragmentCount = 1
0x00, 0x01</td></tr>
<tr><td colspan="32">SessionID =
0x00, 0x00, 0x00, 0x01
0x80, 0x00, 0x00, 0x01</td></tr>
<tr><td colspan="32">ChannelID = 0
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="16">EndAdditionalHeaders = 0x00, 0x00</td><td colspan="16">ConnectionMode = ECDHE
0x00, 0x01</td></tr>
<tr><td colspan="8">ECDHE::MessageType = DeviceAuthenticationRequest
0x02</td><td colspan="16">DeviceCertLength = 387
0x01, 0x83</td><td colspan="8"></td></tr>
<tr><td colspan="32">DeviceCert =
0x30, 0x82, 0x01, 0x7F,
0x30, 0x82, 0x01, 0x26</td></tr>
<tr><td colspan="32">...</td></tr>
</table>

<table>
<tr><td colspan="16">SignedThumbprintLength = 64
0x00, 0x40</td><td colspan="16"></td></tr>
<tr><td colspan="32">SignedThumbprint =
0x1D, 0xDE, 0x16, 0xE0,
0x40, 0xBC, 0x5C, 0xBC</td></tr>
<tr><td colspan="32">...</td></tr>
</table>

4.2.4 Device Authentication Response

Length = 501 bytes.

<table>
<tr><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>20</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>30</th><th>1</th></tr>
<tr><td colspan="16">Signature = 0x30, 0x30</td><td colspan="16">MessageLength = 501 bytes
0x01, 0xF5</td></tr>
<tr><td colspan="8">Version = 0x03</td><td colspan="8">MessageType = Connect
0x02</td><td colspan="16">MessageFlags = None
0x00, 0x00</td></tr>
<tr><td colspan="32">SequenceNumber = 0
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="32">RequestID = 0
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="16">FragmentNumber = 0
0x00, 0x00</td><td colspan="16">FragmentCount = 1
0x00, 0x01</td></tr>
<tr><td colspan="32">SessionID =
0x00, 0x00, 0x00, 0x01
0x80, 0x00, 0x00, 0x01</td></tr>
</table>

<table>
<tr><td colspan="32">ChannelID = 0
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="16">EndAdditionalHeaders = 0x00, 0x00</td><td colspan="16">ConnectionMode = ECDHE
0x00, 0x01</td></tr>
<tr><td colspan="8">ECDHE::MessageType = DeviceAuthenticationResponse
0x02</td><td colspan="16">DeviceCertLength = 388
0x01, 0x84</td><td colspan="8"></td></tr>
<tr><td colspan="32">DeviceCert =
0x30, 0x82, 0x01, 0x80,
0x30, 0x82, 0x01, 0x26</td></tr>
<tr><td colspan="32">...</td></tr>
<tr><td colspan="16">SignedThumbprintLength = 64
0x00, 0x40</td><td colspan="16"></td></tr>
<tr><td colspan="32">SignedThumbprint =
0xC9, 0x5B, 0x87, 0x28,
0xDB, 0x23, 0xF4, 0x23</td></tr>
<tr><td colspan="32">...</td></tr>
</table>

4.2.5 User Device Authentication Request

Length = 422 bytes

<table>
<tr><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>20</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>30</th><th>1</th></tr>
<tr><td colspan="16">Signature = 0x30, 0x30</td><td colspan="16">MessageLength = 422 bytes
0x01, 0xA6</td></tr>
<tr><td colspan="8">Version = 0x03</td><td colspan="8">MessageType = Connect
0x02</td><td colspan="16">MessageFlags = None
0x00, 0x00</td></tr>
</table>

<table>
<tr><td colspan="4">SequenceNumber = 0
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="4">RequestID = 0
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="2">FragmentNumber = 0
0x00, 0x00</td><td colspan="2">FragmentCount = 1
0x00, 0x01</td></tr>
<tr><td colspan="4">SessionID =
0x00, 0x00, 0x00, 0x01
0x00, 0x00, 0x00, 0x01</td></tr>
<tr><td colspan="4">ChannelID = 0
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="2">EndAdditionalHeaders = 0x00, 0x00</td><td colspan="2">ConnectionMode = ECDHE
0x00, 0x01</td></tr>
<tr><td>ECDHE::MessageType = UserDeviceAuthentication Request
0x04</td><td colspan="2">UserDeviceCertLength = 309
0x01, 0x35</td><td></td></tr>
<tr><td colspan="4">UserDeviceCert =
0x30, 0x82, 0x01, 0x31,
0x30, 0x81, 0xD8, 0xA0</td></tr>
<tr><td colspan="4">...</td></tr>
<tr><td colspan="2">SignedThumbprintLength = 64
0x00, 0x40</td><td colspan="2"></td></tr>
<tr><td colspan="4">SignedThumbprint =
0xC9, 0x5B, 0x87, 0x28,
0xDB, 0x23, 0xF4, 0x23</td></tr>
</table>

<table><tr><td colspan="32">...</td></tr></table>

4.2.6 User Device Authentication Response

Length = 421 bytes

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>1
0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>2
0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>3
0</td><td>1</td></tr>
<tr><td colspan="16">Signature = 0x30, 0x30</td><td colspan="16">MessageLength = 421 bytes
0x01, 0xA5</td></tr>
<tr><td colspan="8">Version = 0x03</td><td colspan="8">MessageType = Connect
0x02</td><td colspan="16">MessageFlags = None
0x00, 0x00</td></tr>
<tr><td colspan="32">SequenceNumber = 0
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="32">RequestID = 0
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="16">FragmentNumber = 0
0x00, 0x00</td><td colspan="16">FragmentCount = 1
0x00, 0x01</td></tr>
<tr><td colspan="32">SessionID =
0x00, 0x00, 0x00, 0x01
0x00, 0x00, 0x00, 0x01</td></tr>
<tr><td colspan="32">ChannelID = 0
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="16">EndAdditionalHeaders = 0x00, 0x00</td><td colspan="16">ConnectionMode = ECDHE
0x00, 0x01</td></tr>
<tr><td colspan="8">ECDHE::MessageType = UserDeviceAuthentication Response
0x05</td><td colspan="16">UserDeviceCertLength = 308
0x01, 0x34</td><td colspan="8"></td></tr>
</table>

<table>
<tr><td colspan="2">UserDeviceCert =
0x30, 0x82, 0x01, 0x30,
0x30, 0x81, 0xD8, 0xA0</td></tr>
<tr><td colspan="2">...</td></tr>
<tr><td>SignedThumbprintLength = 64
0x00, 0x40</td><td></td></tr>
<tr><td colspan="2">SignedThumbprint =
0x38, 0x61, 0xE3, 0xCC,
0x24, 0x82, 0x02, 0xCA</td></tr>
<tr><td colspan="2">...</td></tr>
</table>

4.2.7 Authentication Done Request

Length = 45 bytes.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>1
0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>2
0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>3
0</td><td>1</td></tr>
<tr><td colspan="16">Signature = 0x30, 0x30</td><td colspan="16">MessageLength = 45 bytes
0x00, 0x2D</td></tr>
<tr><td colspan="8">Version = 0x03</td><td colspan="8">MessageType = Connect
0x02</td><td colspan="16">MessageFlags = None
0x00, 0x00</td></tr>
<tr><td colspan="32">SequenceNumber = 0
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="32">RequestID = 0
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="16">FragmentNumber = 0
0x00, 0x00</td><td colspan="16">FragmentCount = 1
0x00, 0x01</td></tr>
</table>

<table>
<tr><td colspan="4">SessionID =
0x00, 0x00, 0x00, 0x01
0x00, 0x00, 0x00, 0x01</td></tr>
<tr><td colspan="4">ChannelID = 0
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="2">EndAdditionalHeaders = 0x00, 0x00</td><td colspan="2">ConnectionMode = ECDHE
0x00, 0x01</td></tr>
<tr><td>ECDHE::MessageType = AuthenticationDoneRequest
0x06</td><td colspan="3"></td></tr>
</table>

4.2.8 Authentication Done Response

Length = 46 bytes.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>20</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>30</td><td>1</td></tr>
<tr><td colspan="16">Signature = 0x30, 0x30</td><td colspan="16">MessageLength = 46 bytes
0x00, 0x2E</td></tr>
<tr><td colspan="8">Version = 0x03</td><td colspan="8">MessageType = Connect
0x02</td><td colspan="16">MessageFlags = None
0x00, 0x00</td></tr>
<tr><td colspan="32">SequenceNumber = 0
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="32">RequestID = 0
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="16">FragmentNumber = 0
0x00, 0x00</td><td colspan="16">FragmentCount = 1
0x00, 0x01</td></tr>
</table>

<table>
<tr><td colspan="4">SessionID =
0x00, 0x00, 0x00, 0x01
0x80, 0x00, 0x00, 0x01</td></tr>
<tr><td colspan="4">ChannelID = 0
0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x00, 0x00</td></tr>
<tr><td colspan="2">EndAdditionalHeaders = 0x00, 0x00</td><td colspan="2">ConnectionMode = ECDHE
0x00, 0x01</td></tr>
<tr><td>ECDHE::MessageType = AuthenticationDoneResponse
0x07</td><td>ConnectRequestResult = Success
0x00</td><td></td><td></td></tr>
</table>

5 Security

5.1 Security Considerations for Implementers

None.

5.2 Index of Security Parameters

None.

6 Appendix A: Product Behavior

The information in this specification is applicable to the following Microsoft products or supplemental software. References to product versions include released service packs.

- Windows 10 v1607 operating system
- Windows Server 2016 operating system

Exceptions, if any, are noted below. If a service pack or Quick Fix Engineering (QFE) number appears with the product version, behavior changed in that service pack or QFE. The new behavior also applies to subsequent service packs of the product unless otherwise specified. If a product edition appears with the product version, behavior is different in that product edition.

Unless otherwise specified, any statement of optional behavior in this specification that is prescribed using the terms SHOULD or SHOULD NOT implies product behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term MAY implies that the product does not follow the prescription.

7 Change Tracking

No table of changes is available. The document is either new or has had no changes since its last release.

8 Index