Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: December 3, 2020

T. Aura
Aalto University
M. Sethi
Ericsson

June 1, 2020

Nimble out-of-band authentication for EAP (EAP-NOOB) draft-ietf-emu-eap-noob-01

Abstract

The Extensible Authentication Protocol (EAP) provides support for multiple authentication methods. This document defines the EAP-NOOB authentication method for nimble out-of-band (OOB) authentication and key derivation. The EAP method is intended for bootstrapping all kinds of Internet-of-Things (IoT) devices that have no pre-configured authentication credentials. The method makes use of a user-assisted one-directional OOB message between the peer device and authentication server to authenticate the in-band key exchange. The device must have an input or output interface, such as a display, microphone, speakers or blinking light, which can send or receive dynamically generated messages of tens of bytes in length.

Status of This Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on December 3, 2020.

Copyright Notice

Table of Contents

1. Introduction 3
2. Terminology 4
3. EAP-NOOB protocol 5
3.1. Protocol overview 5
3.2. Protocol messages and sequences 8
3.2.1. Common handshake in all EAP-NOOB exchanges 8
3.2.2. Initial Exchange 10
3.2.3. OOB Step 11
3.2.4. Completion Exchange 13
3.2.5. Waiting Exchange 15
3.3. Protocol data fields 16
3.3.1. Peer identifier, realm and NAI 16
3.3.2. Message data fields 18
3.4. Fast reconnect and rekeying 23
3.4.1. Persistent EAP-NOOB association 23
3.4.2. Reconnect Exchange 24
3.4.3. User reset 27
3.5. Key derivation 28
3.6. Error handling 31
3.6.1. Invalid messages 33
3.6.2. Unwanted peer 33
3.6.3. State mismatch 33
3.6.4. Negotiation failure 33
3.6.5. Cryptographic verification failure 34
3.6.6. Application-specific failure 34
4. IANA Considerations 35
4.1. Cryptosuites 35
4.2. Message Types 36
4.3. Error codes 36
4.4. Domain name reservation considerations 37
5. Implementation Status 38
5.1. Implementation with wpa_supplicant and hostapd 38
5.2. Implementation on Contiki 39
5.3. Protocol modeling 39
6. Security considerations 39
6.1. Authentication principle 39
6.2. Identifying correct endpoints 41
6.3. Trusted path issues and misbinding attacks 42

6.4. Peer identifiers and attributes 43
6.5. Identity protection 44
6.6. Downgrading threats 44
6.7. Recovery from loss of last message 45
6.8. EAP security claims 46
7. References 48
7.1. Normative references 48
7.2. Informative references 49
Appendix A. Exchanges and events per state 51
Appendix B. Application-specific parameters 52
Appendix C. ServerInfo and PeerInfo contents 53
Appendix D. EAP-NOOB roaming 55
Appendix E. OOB message as URL 56
Appendix F. Example messages 57
Appendix G. TODO list 59
Appendix H. Version history 59
Appendix I. Acknowledgments 62
Authors' Addresses 62

1. Introduction

This document describes a method for registration, authentication and key derivation for network-connected ubiquitous computing devices, such as consumer and enterprise appliances that are part of the Internet of Things (IoT). These devices may be off-the-shelf hardware that is sold and distributed without any prior registration or credential-provisioning process, or they may be recycled devices after a hard reset. Thus, the device registration in a server database, ownership of the device, and the authentication credentials for both network access and application-level security must all be established at the time of the device deployment. Furthermore, many such devices have only limited user interfaces that could be used for their configuration. Often, the interfaces are limited to either only input (e.g., camera) or output (e.g., display screen). The device configuration is made more challenging by the fact that the devices may exist in large numbers and may have to be deployed or re-configured nimbly based on user needs.

Commented [DT1]: FYI RFC editor follows Chicago Manual of Style which always puts a comma after “e.g.” and “i.e.”

To summarize, devices may have the following characteristics:

- no pre-established relation with the intended server or user,
- no pre-provisioned device identifier or authentication credentials,
- input or output interface that may be capable of only one-directional communication.

Many proprietary OOB configuration methods exist for specific IoT devices. The goal of this specification is to provide an open standard and a generic protocol for bootstrapping the security of network-connected appliances, such as displays, printers, speakers, and cameras. The security bootstrapping in this specification makes use of a user-assisted out-of-band (OOB) channel. The device authentication relies on a user having physical access to the device, and the key exchange security is based on the assumption that attackers are not able to observe or modify the messages conveyed through the OOB channel. We follow the common approach taken in pairing protocols: performing a Diffie-Hellman key exchange over the insecure network and authenticating the established key with the help of the OOB channel in order to prevent impersonation and man-in-the-middle (MitM) attacks.

The solution presented here is intended for devices that have either an input or output interface, such as a camera, microphone, display screen, speakers or blinking LED light, which is able to send or receive dynamically generated messages of tens of bytes in length. Naturally, this solution may not be appropriate for very small sensors or actuators that have no user interface at all or for devices that are inaccessible to the user. We also assume that the OOB channel is at least partly automated (e.g., camera scanning a bar code) and, thus, there is no need to absolutely minimize the length of the data transferred through the OOB channel. This differs, for example, from Bluetooth simple pairing [BluetoothPairing], where it is critical to minimize the length of the manually transferred or compared codes. Since the OOB messages are dynamically generated, we do not support static printed registration codes. This also prevents attacks where a static secret code would be leaked.

Commented [DT2]: Ok, but can you state the *rationale* for not supporting things like QR code stickers? E.g., does the WG believe that such things are insecure (or at least less secure)?
The “also” in the next sentence implies that sentence is not the rationale, just a side effect, so I assume there must be something more to say.

2. Terminology

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

In addition, this document frequently uses the following terms as they have been defined in [RFC5216]:

authenticator The entity initiating EAP authentication.

peer The entity that responds to the authenticator. In [IEEE-802.1X], this entity is known as the supplicant. (We use the terms peer, device and peer device interchangeably.)

Commented [DT3]: Odd. Why can’t you just pick one and use it everywhere but in this sentence?

server The entity that terminates the EAP authentication method with the peer. In the case where no backend authentication server

is used, the EAP server is part of the authenticator. In the case where the authenticator operates in pass-through mode, the EAP server is located on the backend authentication server.

3. EAP-NOOB protocol

This section defines the EAP-NOOB protocol. The protocol is a generalized version of the original idea presented by Sethi et al. [Sethi14].

3.1. Protocol overview

One EAP-NOOB protocol execution spans two or more EAP conversations, called Exchanges in this specification. Each Exchange consists of several EAP request-response pairs. At least two separate EAP conversations are needed to give the human user time to deliver the OOB message between them.

The overall protocol starts with the Initial Exchange, which comprises four EAP request-response pairs. In the Initial Exchange, the server allocates an identifier to the peer, and the server and peer negotiate the protocol version and cryptosuite (i.e., cryptographic algorithm suite), exchange nonces and perform an Ephemeral Elliptic Curve Diffie-Hellman (ECDHE) key exchange. The user-assisted OOB Step then takes place. This step requires only one out-of-band message either from the peer to the server or from the server to the peer. While waiting for the OOB Step action, the peer MAY probe the server by reconnecting to it with EAP-NOOB. If the OOB Step has already taken place, the probe leads to the Completion Exchange, which completes the mutual authentication and key confirmation. On the other hand, if the OOB Step has not yet taken place, the probe leads to the Waiting Exchange, and the peer will perform another probe after a server-defined minimum waiting time. The Initial Exchange and Waiting Exchange always end in EAP-Failure, while the Completion Exchange may result in EAP-Success. Once the peer and server have performed a successful Completion Exchange, both endpoints store the created association in persistent storage, and the OOB Step is not repeated. Thereafter, creation of new temporal keys, ECDHE rekeying, and updates of cryptographic algorithms can be achieved with the Reconnect Exchange.

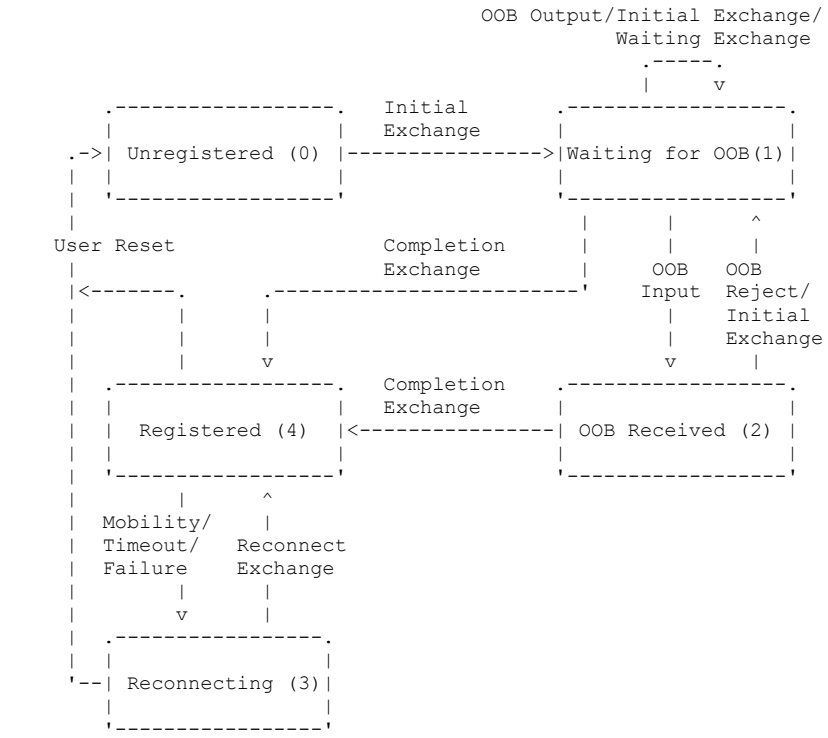

Figure 1: EAP-NOOB server-peer association state machine

Figure 1 shows the association state machine, which is the same for the server and for the peer. (For readability, only the main state transitions are shown. The complete table of transitions can be found in Appendix A.) When the peer initiates the EAP-NOOB method, the server chooses the ensuing message exchange based on the combination of the server and peer states. The EAP server and peer are initially in the Unregistered state, in which no state information needs to be stored. Before a successful Completion Exchange, the server-peer association state is ephemeral in both the server and peer (ephemeral states 0..2), and either endpoint may cause the protocol to fall back to the Initial Exchange. After the Completion Exchange has resulted in EAP-Success, the association

Commented [DT4]: Clarify. Meaning "goes away after a timeout or other type of garbage collection"?

Commented [DT5]: This term is undefined at this point in the doc, at least provide a reference to a section where it is defined.

Commented [DT6]: Do you mean "…fall back to the Unregistered state"? Or did you really mean to use an Initial Exchange without changing state?

state becomes persistent (persistent states 3..4). Only user reset or memory failure can cause the return of the server or the peer from the persistent states to the ephemeral states and to the Initial Exchange.

The server MUST NOT repeat a successful OOB Step with the same peer except if the association with the peer is explicitly reset by the user or lost due to failure of the persistent storage in the server. More specifically, once the association has entered the Registered state, the server MUST NOT delete the association or go back to states 0..2 without explicit user approval. Similarly, the peer MUST NOT repeat the OOB Step unless the user explicitly deletes from the peer the association with the server or resets the peer to the Unregistered state. The server and peer MAY implement user reset of the association by deleting the state data from that endpoint. If an endpoint continues to store data about the association after the user reset, its behavior SHOULD be equivalent to having deleted the association data.

It can happen that the peer accidentally or through user reset loses its persistent state and reconnects to the server without a previously allocated peer identifier. In that case, the server MUST treat the peer as a new peer. The server MAY use auxiliary information, such as the PeerInfo field received in the Initial Exchange, to detect multiple associations with the same peer. However, it MUST NOT delete or merge redundant associations without user or application approval because EAP-NOOB internally has no secure way of verifying that the two peers are the same physical device. Similarly, the server might lose the association state because of a memory failure or user reset. In that case, the only way to recover is that the user also resets the peer.

A special feature of the EAP-NOOB method is that the server is not assumed to have any a-priori knowledge of the peer. Therefore, the peer initially uses the generic identity string "noob@eap-noob.net" (.arpa domain TBA) as its network access identifier (NAI). The server then allocates a server-specific identifier to the peer. The generic NAI serves two purposes: firstly, it tells the server that the peer supports and expects the EAP-NOOB method and, secondly, it allows routing of the EAP-NOOB sessions to a specific authentication server in the AAA architecture.

EAP-NOOB is an unusual EAP method in that the peer has to have multiple EAP conversations with the server before it can receive EAP-Success. The reason is that, while EAP allows delays between the request-response pairs, e.g., for repeated password entry, the user delays in OOB authentication can be much longer than in password trials. In particular, EAP-NOOB also supports ~~also~~ peers with no input

Commented [DT7]: Which states in the diagram are "the ephemeral states"? Unregistered and Waiting for OOB? Clarify. (Appendix A has the answer but the text here is confusing since Appendix A is much later and only an "appendix")

Commented [DT8]: "ephemeral states 0..2" is used above, here just "states 0..2". Suggest using consistent terms throughout.

Commented [DT9]: As phrased, there are 3 compliant behaviors:
1)Follow the MAY and delete the data
2)Don't follow the MAY, do follow the If, and follow the SHOULD and act as if data was deleted
3)Don't follow the MAY, do follow the If, and don't follow the SHOULD and hence act in any other way. In my experience, such undefined behavior can cause interoperability issues.

Is there any reason to permit behavior #3? Why can't you make this be a MUST (since the If already makes it conditional on implementations that don't follow the MAY).

Commented [DT10]: Per https://www.rfc-editor.org/materials/abbrev.expansion.txt this must be expanded on first use since it has no * but it on that page

capability in the user interface. Since users cannot initiate the protocol in these devices, they have to perform the Initial Exchange opportunistically and hope for the OOB Step to take place within a timeout period (NoobTimeout), which is why the timeout needs to be several minutes rather than seconds. For example, consider a printer (peer) that outputs the OOB message on paper, which is then scanned for the server. To support such high-latency OOB channels, the peer and server perform the Initial Exchange in one EAP conversation, then allow time for the OOB message to be delivered, and later perform the Waiting and Completion Exchanges in different EAP conversations.

Commented [DT11]: (Fix grammar)

Commented [DT12]: Elaborate on this use case… Is the assumption that the printer would automatically consume paper in response to any unauthenticated message (hence being a resource consumption attack)? Or that it would only do so *after* getting approval from a human to output the OOB message?

3.2. Protocol messages and sequences

This section defines the EAP-NOOB exchanges, which correspond to EAP conversations. The exchanges start with a common handshake, which determines the type of the following exchange. The common handshake messages and the subsequent messages for each exchange type are listed in the diagrams below. The diagrams also specify the data members present in each message. Each exchange comprises multiple EAP request-response pairs and ends in either EAP-Failure, indicating that authentication is not (yet) successful, or in EAP-Success.

3.2.1. Common handshake in all EAP-NOOB exchanges

All EAP-NOOB exchanges start with common handshake messages. The handshake starts with the identity request and response that are common to all EAP methods. Their purpose is to enable the AAA architecture to route the EAP conversation to the EAP server and to enable the EAP server to select the EAP method. The handshake then continues with one EAP-NOOB request-response pair in which the server discovers the peer identifier used in EAP-NOOB and the peer state.

In more detail, each EAP-NOOB exchange begins with the authenticator sending an EAP-Request/Identity packet to the peer. From this point on, the EAP conversation occurs between the server and the peer, and the authenticator acts as a pass-through device. The peer responds to the authenticator with an EAP-Response/Identity packet, which contains the network access identifier (NAI). The authenticator, acting as a pass-through device, forwards this response and the following EAP conversation between the peer and the AAA architecture. The AAA architecture routes the conversation to a specific AAA server (called "EAP server" or simply "server" in this specification) based on the realm part of the NAI. The server selects the EAP-NOOB method based on the user part of the NAI, as defined in Section 3.3.1.

After receiving the EAP-Response/Identity message, the server sends the first EAP-NOOB request (Type=1) to the peer, which responds with the peer identifier (PeerId) and state (PeerState) in the range 0..3.

However, the peer SHOULD omit the PeerId from the response (Type=1) when PeerState=0. The server then chooses the EAP-NOOB exchange, i.e., the ensuing message sequence, as explained below. The peer recognizes the exchange based on the message type field (Type) of the next EAP-NOOB request received from the server.

The server MUST determine the exchange type based on the combination of the peer and server states as follows (also summarized in Figure 11). If either the peer or server is in the Unregistered (0) state and the other is in one of the ephemeral states (0..2), the server chooses the Initial Exchange. If one of the peer or server is in the OOB Received (2) state and the other is either in the Waiting for OOB (1) or OOB Received (2) state, the OOB Step has taken place and the server chooses the Completion Exchange. If both the server and peer are in the Waiting for OOB (1) state, the server chooses the Waiting Exchange. If the peer is in the Reconnecting (3) state and the server is in the Registered (4) or Reconnecting (3) state, the server chooses the Reconnect Exchange. All other state combinations are error situations where user action is required, and the server SHOULD indicate such errors to the peer with the error code 2002 (see Section 3.6.3). Note also that the peer MUST NOT initiate EAP-NOOB when the peer is in the Registered (4) state.

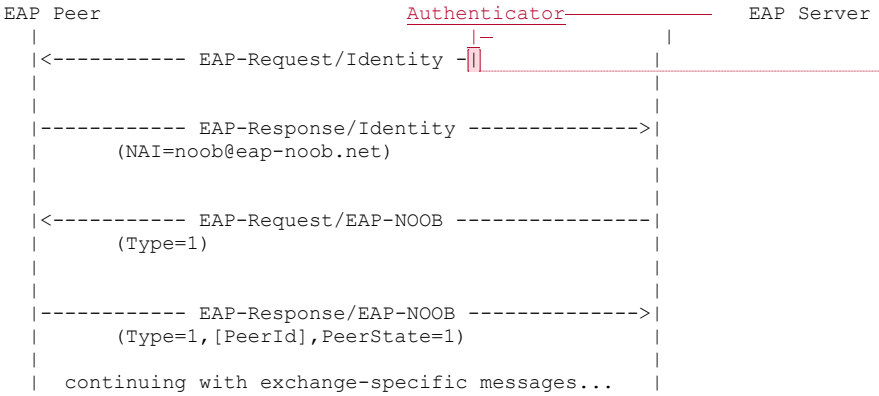

Figure 2: Common handshake in all EAP-NOOB exchanges

Commented [DT13]: Need to label this line (as being the authenticator I assume)

3.2.2. Initial Exchange

The Initial Exchange comprises the common handshake and two further EAP-NOOB request-response pairs, one for version, cryptosuite and parameter negotiation and the other for the ECDHE key exchange. The first EAP-NOOB request (Type=2) from the server contains a newly allocated PeerId for the peer and an optional Realm. The server allocates a new PeerId in the Initial Exchange regardless of any old PeerId in the username part of the received NAI. The server also sends in the request a list of the protocol versions (Vers) and cryptosuites (Cryptosuites) it supports, an indicator of the OOB channel directions it supports (Dirs), and a ServerInfo object. The peer chooses one of the versions and cryptosuites. The peer sends a response (Type=2) with the selected protocol version (Verp), the received PeerId, the selected cryptosuite (Cryptosuitep), an indicator of the OOB channel directions selected by the peer (Dirp), and a PeerInfo object. In the second EAP-NOOB request and response (Type=3), the server and peer exchange the public components of their ECDHE keys and nonces (PKs,Ns,PKp,Np). The ECDHE keys MUST be based on the negotiated cryptosuite, i.e., Cryptosuitep. The Initial Exchange always ends with EAP-Failure from the server because the authentication cannot yet be completed.

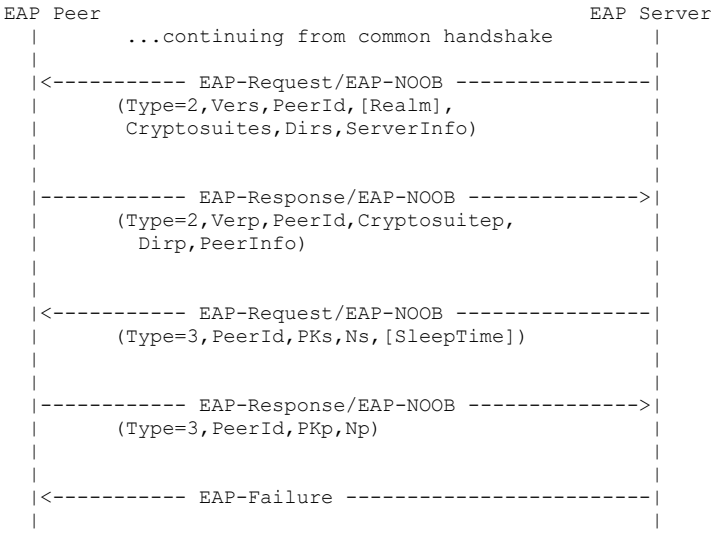

Figure 3: Initial Exchange

At the conclusion of the Initial Exchange, both the server and the peer move to the Waiting for OOB (1) state.

3.2.3. OOB Step

The OOB Step, labeled as OOB Output and OOB Input in Figure 1, takes place after the Initial Exchange. Depending on the negotiated OOB channel direction, the peer or the server outputs the OOB message shown in Figure 4 or Figure 5, respectively. The data fields are the PeerId, the secret nonce Noob, and the cryptographic fingerprint Hoob. The contents of the data fields are defined in Section 3.3.2. The OOB message is delivered to the other endpoint via a user-assisted OOB channel.

For brevity, we will use the terms OOB sender and OOB receiver in addition to the already familiar EAP server and EAP peer. If the OOB message is sent in ~~in~~ the server-to-peer direction, the OOB sender is the server and the OOB receiver is the peer. On the other hand, if the OOB message is sent in the peer-to-server direction, the OOB sender is the peer and the OOB receiver is the server.

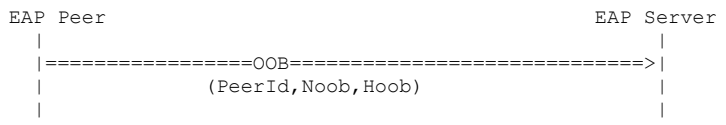

Figure 4: OOB Step, from peer to EAP server

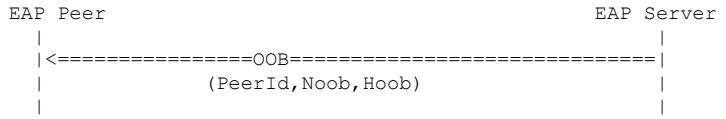

Figure 5: OOB Step, from EAP server to peer

The OOB receiver MUST compare the received value of the fingerprint Hoob (see Section 3.3.2) with a value that it computes locally. This integrity check ensures that the endpoints agree on contents of the Initial Exchange. If the values are equal, the receiver moves to the OOB Received (2) state. Otherwise, the receiver MUST reject the OOB message. For usability reasons, the OOB receiver SHOULD indicate the acceptance or rejection of the OOB message to the user. The receiver SHOULD reject invalid OOB messages without changing its state, until an application-specific number of invalid messages (OobRetries) has been reached, after which the receiver SHOULD consider it an error and go back to the Unregistered (0) state.

The server or peer MAY send multiple OOB messages with different Noob values while in the Waiting for OOB (1) state. The OOB sender SHOULD remember the Noob values until they expire and accept any one of them in the following Completion Exchange. The Noob values sent by the server expire after an application-dependent timeout (NoobTimeout), and the server MUST NOT accept Noob values older than that in the Completion Exchange. The RECOMMENDED value for NoobTimeout is 3600 seconds if there are no application-specific reasons for making it shorter or longer. The Noob values sent by the peer expire as defined in Section 3.2.5.

The OOB receiver does not accept further OOB messages after it has accepted one and moved to the OOB Received (2) state. However, the receiver MAY buffer redundant OOB messages in case an OOB message expiry or similar error detected in the Completion Exchange causes it to return to the Waiting for OOB (1) state. It is RECOMMENDED that

Commented [DT14]: For constrained peer devices, I assume it is acceptable to only remember the most recently received one (since this is only a SHOULD), correct? If so, this text is good.

I see section 6.4 says “User reset or failure in the OOB Step can cause the peer to perform many Initial Exchanges with the server, to allocate many PeerId values, and to store the ephemeral protocol state for them. The peer will typically only remember the latest one.”
I guess that is the answer to my question, and not a different topic, or is the discussion about remembering Noob values and remembering PeerId values two different things?

the OOB receiver notifies the user about redundant OOB messages, but it MAY ~~also~~ instead discard them silently.

The sender will typically generate a new Noob, and therefore a new OOB message, at constant time intervals (NoobInterval). The RECOMMENDED interval is NoobInterval = NoobTimeout / 2, in which case the receiver of the OOB will at any given time accept either of the two latest Noob values. However, the timing of the Noob generation may also be based on user interaction or on implementation considerations.

Even though not recommended (see Section 3.3), this specification allows both directions to be negotiated (Dirp=3) for the OOB channel. In that case, both sides SHOULD output the OOB message, and it is up to the user to deliver (at least) one of them.

The details of the OOB channel implementation including the message encoding are defined by the application. Appendix E gives an example of how the OOB message can be encoded as a URL that may be embedded in a QR code ~~and~~ or NFC tag.

3.2.4. Completion Exchange

After the Initial Exchange, if both the server and the peer support the peer-to-server direction for the OOB channel, the peer SHOULD initiate the EAP-NOOB method again after an applications-specific waiting time in order to probe for completion of the OOB Step. Also, if both sides support the server-to-peer direction of the OOB exchange and the peer receives the OOB message, it SHOULD initiate the EAP-NOOB method immediately. Depending on the combination of the peer and server states, the server continues with ~~with~~ the Completion Exchange or Waiting Exchange (see Section 3.2.1 on how the server makes this decision).

The Completion Exchange comprises the common handshake and one or two further EAP-NOOB request-response pairs. If the peer is in the Waiting for OOB (1) state, the OOB message has been sent in the peer-to-server direction. In that case, only one request-response pair (Type=6) takes place. In the request, the server sends the NoobId value, which the peer uses to identify the exact OOB message received by the server. On the other hand, if the peer is in the OOB Received (2) state, the direction of the OOB message is from server to peer. In this case, two request-response pairs (Type=5 and Type=6) are needed. The purpose of the first request-response pair (Type=5) is that it enables the server to discover NoobId, which identifies the exact OOB message received by the peer. The server returns the same NoobId to the peer in the latter request.

Commented [DT15]: (Discarding silently is instead of, not in addition to, notifying the user)

Commented [DT16]: "(at least) one"? Or *only* one? The last paragraph on page 14 seems to show that the protocol is robust enough to handle the case where both are delivered, so "(at least) one" seems the intent here.

Commented [DT17]: ("or", to match language in Appendix E)

In the last request-response pair (Type=6) of the Completion Exchange, the server and peer exchange message authentication codes. Both sides MUST compute the keys Kms and Kmp as defined in Section 3.5 and the message authentication codes MACs and MACp as defined in Section 3.3.2. Both sides MUST compare the received message authentication code with a locally computed value. If the peer finds that it has received the correct value of MACs and the server finds that it has received the correct value of MACp, the Completion Exchange ends in EAP-Success. Otherwise, the endpoint where the comparison fails indicates this with an error message (error code 4001, see Section 3.6.1) and the Completion Exchange ends in EAP-Failure.

After successful Completion Exchange, both the server and the peer move to the Registered (4) state. They also derive the output keying material and store the persistent EAP-NOOB association state as defined in Section 3.4 and Section 3.5.

It is possible that the OOB message expires before it is received. In that case, the sender of the OOB message no longer recognizes the NoobId that it receives in the Completion Exchange. Another reason why the OOB sender might not recognize the NoobId is if the received OOB message was spoofed and contained an attacker-generated Noob value. The recipient of an unrecognized NoobId indicates this with an error message (error code 2003, see Section 3.6.1) and the Completion Exchange ends in EAP-Failure. The recipient of the error message 2003 moves back to the Waiting for OOB (1) state. This state transition is shown as OOB Reject in Figure 1 (even though it really is a specific type of failed Completion Exchange). The sender of the error message, on the other hand, stays in its previous state.

Although it is not expected to occur in practice, poor user interface design could lead to two OOB messages delivered simultaneously, one from the peer to the server and the other from the server to the peer. The server detects this event in the beginning of the Completion Exchange by observing that both the server and peer are in the OOB Received state (2). In that case, as a tiebreaker, the server MUST behave as if only the server-to-peer message had been delivered.

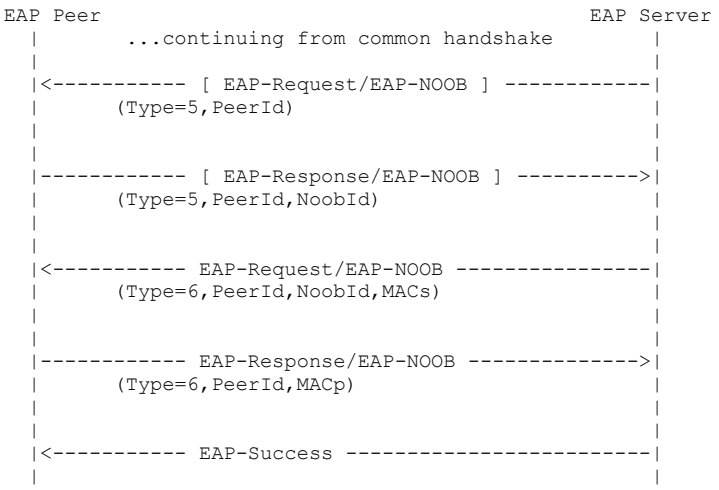

Figure 6: Completion Exchange

3.2.5. Waiting Exchange

As explained in Section 3.2.4, the peer SHOULD probe the server for completion of the OOB Step. When the combination of the peer and server states indicates that the OOB message has not yet been delivered, the server chooses the Waiting Exchange (see Section 3.2.1 on how the server makes this decision). The Waiting Exchange comprises the common handshake and one further request-response pair, and it ends always in EAP-Failure.

In order to limit the rate at which peers probe the server, the server MAY send to the peer either in the Initial Exchange or in the Waiting Exchange a minimum time to wait before probing the server again. A peer that has not received an OOB message MUST wait at least the server-specified minimum waiting time in seconds (SleepTime) before initiating EAP again with the same server. The peer uses the latest SleepTime value that it has received in or after the Initial Exchange. If the server has not sent any SleepTime value, the peer SHOULD wait for an application-specified minimum time (SleepTimeDefault).

After the Waiting Exchange, the peer MUST discard (from its local ephemeral storage) Noob values that it has sent to the server in OOB

Commented [DT18]: This means that this protocol cannot be easily implemented in IoT devices that have no relative time clock. Does EAP itself already have this limitation regardless of EAP method? If not, it would be good to call this out, since this limits applicability to constrained devices.

Commented [DT19]: Since the minimum time is app-specified as this sentence says, what does it mean to *not* follow this SHOULD? I.e., why is it not a MUST?

messages that are older than the application-defined timeout NoobTimeout (see Section 3.2.3). The peer SHOULD discard such expired Noob values even if the probing failed, e.g., because of failure to connect to the EAP server or incorrect HMAC. The timeout of peer-generated Noob values is defined like this in order to allow the peer to probe the server once after it has waited for the server-specified SleepTime.

Commented [DT20]: Another acronym that needs expansion according to the RFC editor style guide

If the server and peer have negotiated to use only the server-to-peer direction for the OOB channel (Dirp=2), the peer SHOULD nevertheless probe the server. The purpose of this is to keep the server informed about the peers that are still waiting for OOB messages. The server MAY set SleepTime to a high number (3600) to prevent the peer from probing the server frequently.

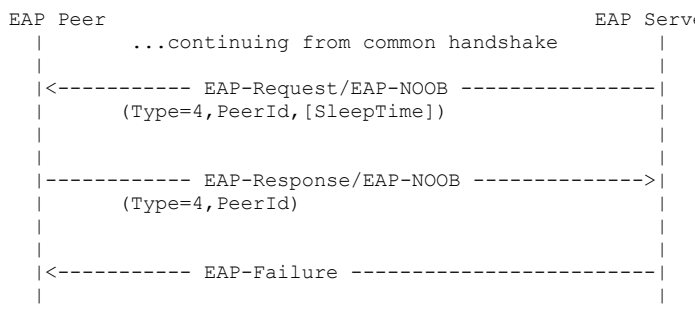

Figure 7: Waiting Exchange

3.3. Protocol data fields

This section defines the various identifiers and data fields used in the EAP-NOOB protocol.

3.3.1. Peer identifier, realm and NAI

The server allocates a new peer identifier (PeerId) for the peer in the Initial Exchange. The peer identifier MUST follow the syntax of the utf8-username specified in [RFC7542]. The server MUST generate the identifiers in such a way that they do not repeat and cannot be guessed by the peer or third parties before the server sends them to the peer in the Initial Exchange. One way to generate the identifiers is to choose a random 16-byte identifier and to base64url encode it without padding [RFC4648] into a 22-character string.

Another way to generate the identifiers is to choose a random 22-character alphanumeric string. It is RECOMMENDED to not use identifiers longer than this because they result in longer OOB messages.

The peer uses the allocated PeerId to identify itself to the server in the subsequent exchanges. It sets the PeerId value in response type 1 as follows. When the peer is in the Unregistered (0) state, it SHOULD omit the PeerId from response type 1. When the peer is in one of the states 1..2, it MUST use the PeerId that the server assigned to it in the latest Initial Exchange. When the peer is in one of the persistent states 3..4, it MUST use the PeerId from its persistent EAP-NOOB association. (The PeerId is written to the association when the peer moves to the Registered (4) state after a Completion Exchange.)

The default realm for the peer is "eap-noob.net" (.arpa domain TBA). However, the user or application MAY provide a different default realm to the peer. Furthermore, the server MAY assign a new realm to the peer in the Initial Exchange or Reconnect Exchange, in the Realm field of response types 2 and 7. The Realm value MUST follow the syntax of the utf8-realm specified in [RFC7542]. When the peer is in the Unregistered (0) state, or when the peer is in one of the states 1..2 and the server did not send a Realm in the latest Initial Exchange, the peer MUST use the default realm. When the peer is in one of the states 1..2 and the server sent a Realm in the latest Initial Exchange, the peer MUST use that realm. Finally, when the peer is in one of the persistent states 3..4, it MUST use the Realm from its persistent EAP-NOOB association. (The Realm is written to the association when the peer moves to the Registered (4) state after a Completion Exchange or Reconnect Exchange.)

Commented [DT21]: Humans don't implement MAYs in this sense. Suggest
"However, the peer MAY allow the user or application to provide a different default realm."
such that the MAY now applies to the entity implementing this spec (i.e., the peer).

To compose its NAI [RFC7542], the peer concatenates the string "noob@" and the server-assigned realm. When no server-assigned realm is available, the default value is used instead.

The purpose of the server-assigned realm is to enable more flexible routing of the EAP sessions over the AAA infrastructure, including roaming scenarios (see Appendix D). Moreover, some Authenticators or AAA servers use the assigned Realm to determine peer-specific connection parameters, such as isolating the peer to a specific VLAN. The possibility to configure a different default realm enables registration of new devices while roaming. It also enables manufacturers to set up their own AAA servers for bootstrapping of new peer devices.

The peer's PeerId and Realm are ephemeral until a successful Completion Exchange takes place. Thereafter, the values become parts

of the persistent EAP-NOOB association, until the user resets the peer and the server or until a new Realm is assigned in the Reconnect Exchange.

3.3.2. Message data fields

Table 1 defines the data fields in the protocol messages. The in-band messages are formatted as JSON objects [RFC8259] in UTF-8 encoding. The JSON member names are in the left-hand column of the table.

Commented [DT22]: So this limits applicability to constrained IoT devices, since JSON can be verbose compared to, say, CBOR, and if the IoT device already uses CBOR for its normal protocol use this requires adding a separate parser for JSON which may cause code size issues. Is there a rationale for why CBOR could not be an option? E.g., if this protocol is not applicable for constrained devices, then say so. (I don't know whether EAP itself already inherently has problems that limit its applicability for constrained devices.)

Data field	Description
Vers, Verp	EAP-NOOB protocol versions supported by the EAP server, and the protocol version chosen by the peer. Vers is a JSON array of unsigned integers, and Verp is an unsigned integer. Example values are "[1]" and "1", respectively.
PeerId	Peer identifier as defined in Section 3.3.1.
Realm	Peer realm as defined in Section 3.3.1.
PeerState	Peer state is an integer in the range 0..4 (see Figure 1). However, only values 0..3 are ever sent in the protocol messages.
Type	EAP-NOOB message type. The type is an integer in the range 0..9. EAP-NOOB requests and the corresponding responses share the same type value.
PKs, PKp	The public components of the ECDHE keys of the server and peer. PKs and PKp are sent in the JSON Web Key (JWK) format [RFC7517]. The dDetailed format of the JWK object is defined by the cryptosuite.
Cryptosuites, Cryptosuitep	The identifiers of cryptosuites supported by the server and of the cryptosuite selected by the peer. The server-supported cryptosuites in Cryptosuites are formatted as a JSON array of the identifier integers. The server MUST send a nonempty array with no repeating elements, ordered by decreasing priority. The peer MUST respond with exactly one suite in the

	Cryptosuitep value, formatted as an identifier integer. The registration of cryptosuites is specified in Section 4.1. Example values are "[1]" and "1", respectively.
Dirs, Dirp	The OOB channel directions supported by the server and the directions selected by the peer. The possible values are 1=peer-to-server, 2=server-to-peer, 3=both directions.
Dir	The actual direction of the OOB message (1=peer-to-server, 2=server-to-peer). This value is not sent over any communication channel but it is included in the computation of the cryptographic fingerprint Hoob.
Ns, Np	32-byte nonces for the Initial Exchange.
ServerInfo	This field contains information about the server to be passed from the EAP method to the application layer in the peer. The information is specific to the application or to the OOB channel and it is encoded as a JSON object of at most 500 bytes. It could include, for example, the access-network name and server name or a Uniform Resource Locator (URL) [RFC4266] or some other information that helps the user to deliver the OOB message to the server through the out-of-band channel.
PeerInfo	This field contains information about the peer to be passed from the EAP method to the application layer in the server. The information is specific to the application or to the OOB channel and it is encoded as a JSON object of at most 500 bytes. It could include, for example, the peer brand, model and serial number, which help the user to distinguish between devices and to deliver the OOB message to the correct peer through the out-of-band channel.
SleepTime	The number of seconds for which the peer MUST NOT start a new execution of the EAP-NOOB method with the authenticator, unless the peer receives the OOB message or the peer is reset by the user. The server can use this field to limit the rate at which peers probe it.

	SleepTime is an unsigned integer in the range 0..3600.
Noob	16-byte secret nonce sent through the OOB channel and used for the session key derivation. The endpoint that received the OOB message uses this secret in the Completion Exchange to authenticate the exchanged key to the endpoint that sent the OOB message.
Hoob	16-byte cryptographic fingerprint (i.e., hash value) computed from all the parameters exchanged in the Initial Exchange and in the OOB message. Receiving this fingerprint over the OOB channel guarantees the integrity of the key exchange and parameter negotiation. Hence, it authenticates the exchanged key to the endpoint that receives the OOB message.
NoobId	16-byte identifier for the OOB message, computed with a one-way function from the nonce Noob in the message.
MACs, MACp	Message authentication codes (HMAC) for mutual authentication, key confirmation, and integrity check on the exchanged information. The input to the HMAC is defined below, and the key for the HMAC is defined in Section 3.5.
Ns2, Np2	32-byte Nonces for the Reconnect Exchange.
KeyingMode	Integer indicating the key derivation method. 0 in the Completion Exchange, and 1..3 in the Reconnect Exchange.
PKs2, PKp2	The public components of the ECDHE keys of the server and peer for the Reconnect Exchange. PKp2 and PKs2 are sent in the JSON Web Key (JWK) format [RFC7517]. The detailed format of the JWK object is defined by the cryptosuite.
MACs2, MACp2	Message authentication codes (HMAC) for mutual authentication, key confirmation, and integrity check on the Reconnect Exchange. The input to the HMAC is defined below, and the key for the HMAC is defined in Section 3.5.

ErrorCode	Integer indicating an error condition. Defined in Section 4.3.
ErrorInfo	Textual error message for logging and debugging purposes. A UTF-8 string of at most 500 bytes.

Table 1: Message data fields

It is RECOMMENDED for servers to support both OOB channel directions (Dirs=3), unless the type of the OOB channel limits them to one direction (Dirs=1 or Dirs=2). On the other hand, it is RECOMMENDED that the peer selects only one direction (Dirp=1 or Dirp=2) even when both directions (Dirp=3) would be technically possible. The reason is that, if value 3 is negotiated, the user may be presented with two OOB messages, one for each direction, even though only one of them needs to be delivered. This can be confusing to the user. Nevertheless, the EAP-NOOB protocol is designed to cope also with selected value 3, in which case it uses the first delivered OOB message. In the unlikely case of simultaneously delivered OOB messages, the protocol prioritizes the server-to-peer direction.

The nonces in the in-band messages (Ns, Np, Ns2, Np2) are 32-byte fresh random byte strings, and the secret nonce Noob is a 16-byte fresh random byte string. All the nonces are generated by the endpoint that sends the message.

The fingerprint Hoob and the identifier NoobId are computed with the cryptographic hash function specified in the negotiated cryptosuite and truncated to the 16 leftmost bytes of the output. The message authentication codes (MACs, MACp, MACs2, MACp2) are computed with the HMAC function [RFC2104] based on the same cryptographic hash function and truncated to the 32 leftmost bytes of the output.

The inputs to the hash function for computing the fingerprint Hoob and to the HMAC for computing MACs, MACp, MACs2 and MACp2 are JSON arrays containing a fixed number (17) of elements. The array elements MUST be copied to the array verbatim from the sent and received in-band messages. When the element is a JSON object, its members MUST NOT be reordered or re-encoded. Whitespace MUST NOT be added anywhere in the JSON structure. Implementers should check that their JSON library copies the elements as UTF-8 strings and does not modify them in any way, and that it does not add whitespace to the HMAC input.

The inputs for computing the fingerprint and message authentication codes are the following:

```
Hoob = H(Dir,Vers,Verp,PeerId,Cryptosuites,Dirs,ServerInfo,Cryptos
uitep,Dirp,[Realm],PeerInfo,0,PKs,Ns,PKp,Np,Noob).

NoobId = H("NoobId",Noob).

MACs = HMAC(Kms; 2,Vers,Verp,PeerId,Cryptosuites,Dirs,ServerInfo,C
ryptosuitep,Dirp,[Realm],PeerInfo,0,PKs,Ns,PKp,Np,Noob).

MACp = HMAC(Kmp; 1,Vers,Verp,PeerId,Cryptosuites,Dirs,ServerInfo,C
ryptosuitep,Dirp,[Realm],PeerInfo,0,PKs,Ns,PKp,Np,Noob).

MACs2 = HMAC(Kms2; 2,Vers,Verp,PeerId,Cryptosuites,"",[ServerInfo]
,Cryptosuitep,"",[Realm],[PeerInfo],KeyingMode,[PKs2],Ns2,[PKp2],N
p2,"")

MACp2 = HMAC(Kmp2; 1,Vers,Verp,PeerId,Cryptosuites,"",[ServerInfo]
,Cryptosuitep,"",[Realm],[PeerInfo],KeyingMode,[PKs2],Ns2,[PKp2],N
p2,"")
```

The inputs denoted with "" above are not present, and the values in brackets [] are optional. Both kinds of missing input values are represented by empty strings "" in the HMAC input (JSON array). Realm is included in the computation of MACs and MACp if it was sent or received in the preceding Initial Exchange. Each of the values in brackets for the computation of Macs2 and Macp2 MUST be included if it was sent or received in the same Reconnect Exchange; otherwise the value is replaced by an empty string "".

The parameter Dir indicates the direction in which the OOB message containing the Noob value is being sent (1=peer-to-server, 2=server-to-peer). This field is included in the Hoob input to prevent the user from accidentally delivering the OOB message back to its originator in the rare cases where both OOB directions have been negotiated. The keys (Kms, Kmp, Kms2, Kmp2) for the HMACs are defined in Section 3.5.

The nonces (Ns, Np, Ns2, Np2, Noob) and the hash value (NoobId) MUST be base64url encoded [RFC4648] when they are used as input to the cryptographic functions H or HMAC. These values and the message authentication codes (MACs, MACp, MACs2, MACp2) MUST also be base64url encoded when they are sent in the in-band messages. The values Noob and Hoob in the OOB channel MAY be base64url encoded if that is appropriate for the application and the OOB channel. All base64url encoding is done without padding. The base64url encoded values will naturally consume more space than the number of bytes

specified above (22-character string for a 16-byte nonce and 43-character string for a 32-byte nonce or message authentication code). In the key derivation in Section 3.5, on the other hand, the unencoded nonces (raw bytes) are used as input to the key derivation function.

The ServerInfo and PeerInfo are JSON objects with UTF-8 encoding. The length of either encoded object as a byte array MUST NOT exceed 500 bytes. The format and semantics of these objects MUST be defined by the application that uses the EAP-NOOB method.

3.4. Fast reconnect and rekeying

EAP-NOOB implements Fast Reconnect ([RFC3748], section 7.2.1) that avoids repeated use of the user-assisted OOB channel.

The rekeying and the Reconnect Exchange may be needed for several reasons. New EAP output values MSK and EMSK may be needed because of mobility or timeout of session keys. Software or hardware failure or user action may also cause the authenticator, EAP server or peer to lose its non-persistent state data. The failure would typically be detected by the peer or authenticator when session keys are no longer accepted by the other endpoint. Changes in the supported cryptosuites in the EAP server or peer may also cause the need for a new key exchange. When the EAP server or peer detects any one of these events, it MUST change from the Registered to Reconnecting state. These state transitions are labeled Mobility/Timeout/Failure in Figure 1. The EAP-NOOB method will then perform the Reconnect Exchange the next time ~~when~~ that EAP is triggered.

3.4.1. Persistent EAP-NOOB association

To enable rekeying, the EAP server and peer store the session state in persistent memory after a successful Completion Exchange. This state data, called "persistent EAP-NOOB association", MUST include at least the data fields shown in Table 2. They are used for identifying and authenticating the peer in the Reconnect Exchange. When a persistent EAP-NOOB association exists, the EAP server and peer are in the Registered state (4) or Reconnecting state (3), as shown in Figure 1.

Data field	Value	Type
PeerId	Peer identifier allocated by server	UTF-8 string (typically 22 bytes)
Verp	Negotiated protocol version	integer
Cryptosuitep	Negotiated cryptosuite	integer
CryptosuitepPrev (at peer only)	Previous cryptosuite	integer
Realm	Optional realm assigned by server (default value is "eap-noob.net" (.arpa domain TBA))	UTF-8 string
Kz	Persistent key material	32 bytes
KzPrev (at peer only)	Previous Kz value	32 bytes

Table 2: Persistent EAP-NOOB association

Commented [DT23]: Top of page 17 says “Another way to generate the identifiers is to choose a random 22-character alphanumeric string” and “alphanumeric” is defined at https://www.merriam-webster.com/dictionary/alphanumeric as being letters and numbers. Unicode does not restrict “letters” or “numbers” to ascii.

Note that a UTF-8 character can be anywhere from 1 to 4 bytes long. A 22 character string could thus be up to 88 bytes long. The word “typically” means you think ASCII (1 byte UTF-8) is typical.

If you meant ASCII back on page 17 then say so. Otherwise, change this to “characters”.

3.4.2. Reconnect Exchange

The server chooses the Reconnect Exchange when both the peer and the server are in a persistent state and fast reconnection is needed (see Section 3.2.1 for details).

The Reconnect Exchange comprises the common handshake and three further EAP-NOOB request-response pairs, one for cryptosuite and parameter negotiation, another for the nonce and ECDHE key exchange, and the last one for exchanging message authentication codes. In the first request and response (Type=7) the server and peer negotiate a protocol version and cryptosuite in the same way as in the Initial Exchange. The server SHOULD NOT offer and the peer MUST NOT accept protocol versions or cryptosuites that it knows to be weaker than the one currently in the Cryptosuitep field of the persistent EAP-NOOB association. The server SHOULD NOT needlessly change the cryptosuites it offers to the same peer because peer devices may have limited ability to update their persistent storage. However, if the peer has different values in the Cryptosuitep and CryptosuitepPrev fields, it SHOULD also accept offers that are not weaker than CryptosuitepPrev. Note that Cryptosuitep and CryptosuitePrev from the persistent EAP-NOOB association are only used to support the

negotiation as described above; all actual cryptographic operations use the negotiated cryptosuite. The request and response (Type=7) MAY additionally contain PeerInfo and ServerInfo objects.

The server then determines the KeyingMode (defined in Section 3.5) based on changes in the negotiated cryptosuite and whether it desires to achieve forward secrecy or not. The server SHOULD only select KeyingMode 3 when the negotiated cryptosuite differs from the Cryptosuitep in the server's persistent EAP-NOOB association, although it is technically possible to select this values without changing the cryptosuite. In the second request and response (Type=8), the server informs the peer about the KeyingMode, and the server and peer exchange nonces (Ns2, Np2). When KeyingMode is 2 or 3 (rekeying with ECDHE), they also exchange public components of ECDHE keys (PKs2, PKp2). The server ECDHE key MUST be fresh, i.e., not previously used with the same peer, and the peer ECDHE key SHOULD be fresh, i.e., not previously used.

In the third and final request and response (Type=9), the server and peer exchange message authentication codes. Both sides MUST compute the keys Kms2 and Kmp2 as defined in Section 3.5 and the message authentication codes MACs2 and MACp2 as defined in Section 3.3.2. Both sides MUST compare the received message authentication code with a locally computed value.

The rules by which the peer compares the received MACs2 are non-trivial because, in addition to authenticating the current exchange, MACs2 may confirm the success or failure of a recent cryptosuite upgrade. The peer processes the final request (Type=9) as follows:

1. The peer first compares the received MACs2 value with one it computed using the Kz stored in the persistent EAP-NOOB association. If the received and computed values match, the peer deletes any data stored in the CryptosuitepPrev and KzPrev fields of the persistent EAP-NOOB association. It does this because the received MACs2 confirms that the peer and server share the same Cryptosuitep and Kz, and any previous values must no longer be accepted.

2. If, on the other hand, the peer finds that the received MACs2 value does not match the one it computed locally with Kz, the peer checks whether the KzPrev field in the persistent EAP-NOOB association stores a key. If it does, the peer repeats the key derivation (Section 3.5) and local MACs2 computation (Section 3.3.2) using KzPrev in place of Kz. If this second computed MACs2 matches the received value, the match indicates synchronization failure caused by the loss of the last response (Type=9) in a previously attempted cryptosuite upgrade. In this

case, the peer rolls back that upgrade by overwriting Cryptosuitep with CryptosuitepPrev and Kz with KzPrev in the persistent EAP-NOOB association. It also clears the CryptosuitepPrev and KzPrev fields.

3. If the received MACs2 matched one of the locally computed values, the peer proceeds to send the final response (Type=9). The peer also moves to the Registered (4) state. When KeyingMode is 1 or 2, the peer stops here. When KeyingMode is 3, the peer also updates the persistent EAP-NOOB association with the negotiated Cryptosuitep and the newly-derived Kz value. To prepare for possible synchronization failure caused by the loss of the final response (Type=9) during cryptosuite upgrade, the peer copies the old Cryptosuitep and Kz values in the persistent EAP-NOOB association to the CryptosuitepPrev and KzPrev fields.

4. Finally, if the peer finds that the received MACs2 does not match either of the two values that it computed locally (or one value if no KzPrev was stored), the peer sends an error message (error code 4001, see Section 3.6.1), which causes the ~~the~~ Reconnect Exchange to end in EAP-Failure.

The server rules for processing the final message are simpler than the peer rules because the server does not store previous keys and it never rolls back a cryptosuite upgrade. Upon receiving the final response (Type=9), the server compares the received value of MACp2 with one it computes locally. If the values match, the Reconnect Exchange ends in EAP-Success. When KeyingMode is 3, the server also updates Cryptosuitep and Kz in the persistent EAP-NOOB association. On the other hand, if the server finds that the values do not match, it sends an error message (error code 4001), and the Reconnect Exchange ends in EAP-Failure.

The endpoints MAY send updated Realm, ServerInfo and PeerInfo objects in the Reconnect Exchange. When there is no update to the values, they SHOULD omit this information from the messages. If the Realm was sent, each side updates Realm in the persistent EAP-NOOB association when moving to the Registered (4) state.

Commented [DT24]: So if this MAY is not followed, does that mean any updates are not sent at all, or that they are sent via some other mechanism?

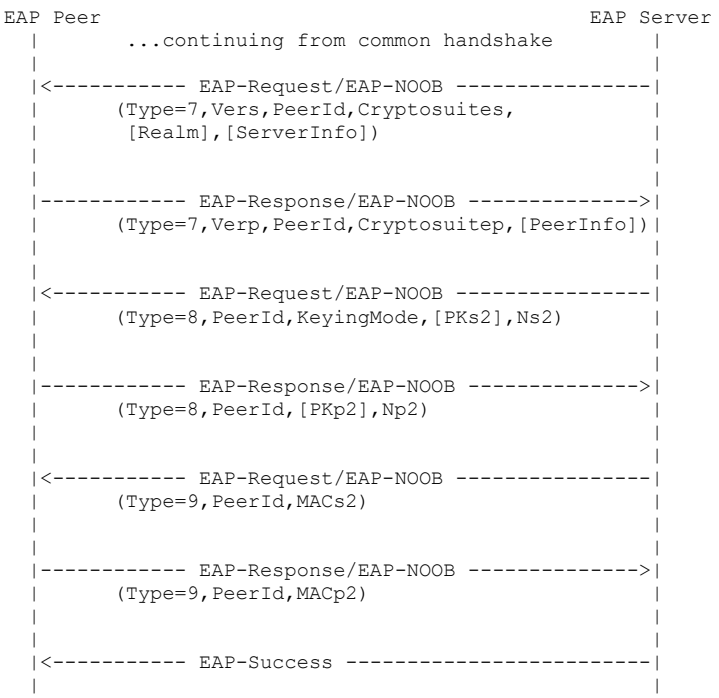

Figure 8: Reconnect Exchange

3.4.3. User reset

As shown in the association state machine in Figure 1, the only specified way for the association to return from the Registered state (4) to the Unregistered state (0) is through user-initiated reset. After the reset, a new OOB message will be needed to establish a new association between the EAP server and peer. Typical situations in which the user reset is required are when the other side has accidentally lost the persistent EAP-NOOB association data, or when the peer device is decommissioned.

The server could detect that the peer is in the Registered or Reconnecting state but the server itself is in one of the ephemeral states 0..2 (including situations where the server does not recognize

Commented [DT25]: The server “recognizing” the PeerId implies that it compares it to existing state, but PeerId is a utf-8 string. Is comparison to be done byte-wise? (I.e., it must match exactly, with no differences in NFC vs NFD, or half-width vs full-width, or case, or punycode-encoded realm vs not, etc.)

If I understand correctly, the peerId is assigned by the server, and sent back to the same server, and does not need to be entered by a human, and so a byte-wise comparison should suffice and you can say that explicitly.

the PeerId). In this case, effort should be made to recover the persistent server state, for example, from a backup storage - especially if many peer devices are similarly affected. If that is not possible, the EAP server SHOULD log the error or notify an administrator. The only way to continue from such a situation is by having the user reset the peer device.

On the other hand, if the peer is in any of the ephemeral states 0..2, including the Unregistered state, the server will treat the peer as a new peer device and allocate a new PeerId to it. The PeerInfo can be used by the user as a clue to which physical device has lost its state. However, there is no secure way of matching the "new" peer with the old PeerId without repeating the OOB Step. This situation will be resolved when the user performs the OOB Step and, thus, identifies the physical peer device. The server user interface MAY support situations where the "new" peer is actually a previously registered peer that has been reset by a user or otherwise lost its persistent data. In those cases, the user could choose to merge the new peer identity with the old one in the server. The alternative is to treat the device just like a new peer.

3.5. Key derivation

EAP-NOOB derives the EAP output values MSK and EMSK and other secret keying material from the output of an Ephemeral Elliptic Curve Diffie-Hellman (ECDHE) algorithm following the NIST specification [NIST-DH]. In NIST terminology, we use a C(2e, 0s, ECC CDH) scheme, i.e., two ephemeral keys and no static keys. In the Initial and Reconnect Exchanges, the server and peer compute the ECDHE shared secret Z as defined in section 6.1.2.2 of the NIST specification [NIST-DH]. In the Completion and Reconnect Exchanges, the server and peer compute the secret keying material from Z with the single-step key derivation function (KDF) defined in section 5.8.1 of the NIST specification. The hash function H for KDF is taken from the negotiated cryptosuite.

KeyingMode	Description
0	Completion Exchange (always with ECDHE)
1	Reconnect Exchange, rekeying without ECDHE
2	Reconnect Exchange, rekeying with ECHDE, no change in cryptosuite
3	Reconnect Exchange, rekeying with ECDHE, new cryptosuite negotiated

Table 3: Keying modes

The key derivation has four different modes (KeyingMode), which are specified in Table 3. Table 4 defines the inputs to KDF in each KeyingMode.

In the Completion Exchange (KeyingMode=0), the input Z comes from the preceding Initial exchange. KDF takes some additional inputs (OtherInfo), for which we use the concatenation format defined in section 5.8.1.2.1 of the NIST specification [NIST-DH]. OtherInfo consists of the AlgorithmId, PartyUInfo, PartyVInfo, and SuppPrivInfo fields. The first three fields are fixed-length bit strings, and SuppPrivInfo is a variable-length string with a one-byte Datalength counter. AlgorithmId is the fixed-length 8-byte ASCII string "EAP-NOOB". The other input values are the server and peer nonces. In the Completion Exchange, the inputs also include the secret nonce Noob from the OOB message.

In the simplest form of the Reconnect Exchange (KeyingMode=1), fresh nonces are exchanged but no ECDHE keys are sent. In this case, input Z to the KDF is replaced with the shared key Kz from the persistent EAP-NOOB association. The result is rekeying without the computational cost of the ECDHE exchange, but also without forward secrecy.

When forward secrecy is desired in the Reconnect Exchange (KeyingMode=2 or KeyingMode=3), both nonces and ECDHE keys are exchanged. Input Z is the fresh shared secret from the ECDHE exchange with PKs2 and PKp2. The inputs also include the shared secret Kz from the persistent EAP-NOOB association. This binds the rekeying output to the previously authenticated keys.

KeyingMode	KDF input field	Value	Length (bytes)
0 Completion	Z	ECDHE shared secret from PKs and PKp	variable
	AlgorithmId	"EAP-NOOB"	8
	PartyUInfo	Np	32
	PartyVInfo	Ns	32
	SuppPubInfo	(not allowed)	
	SuppPrivInfo	Noob	16
1 Reconnect, rekeying without ECDHE	Z	Kz	32
	AlgorithmId	"EAP-NOOB"	8
	PartyUInfo	Np2	32
	PartyVInfo	Ns2	32
	SuppPubInfo	(not allowed)	
	SuppPrivInfo	(null)	0
2 or 3 Reconnect, rekeying, with ECDHE, same or new cryptosuite	Z	ECDHE shared secret from PKs2 and PKp2	variable
	AlgorithmId	"EAP-NOOB"	8
	PartyUInfo	Np2	32
	PartyVInfo	Ns2	32
	SuppPubInfo	(not allowed)	
	SuppPrivInfo	Kz	32

Table 4: Key derivation input

Table 5 defines how the output bytes of KDF are used. In addition to the EAP output values MSK and EMSK, the server and peer derive another shared secret key AMSK, which MAY be used for application-layer security. Further output bytes are used internally by EAP-NOOB for the message authentication keys (Kms, Kmp, Kms2, Kmp2).

The Completion Exchange (KeyingMode=0) produces the shared secret Kz, which the server and peer store in the persistent EAP-NOOB association. When a new cryptosuite is negotiated in the Reconnect Exchange (KeyingMode=3), it similarly produces a new Kz. In that case, the server and peer update both the cryptosuite and Kz in the persistent EAP-NOOB association. Additionally, the peer stores the previous Cryptosuitep and Kz values in the CryptosuitepPrev and KzPrev fields of the persistent EAP-NOOB association.

KeyingMode	KDF output bytes	Used as	Length (bytes)
0	0..63	MSK	64
Completion	64..127	EMSK	64
	128..191	AMSK	64
	192..223	MethodId	32
	224..255	Kms	32
	256..287	Kmp	32
	288..319	Kz	32
1 or 2 Reconnect, rekeying without ECDHE, or with ECDHE and unchanged cryptosuite	0..63	MSK	64
	64..127	EMSK	64
	128..191	AMSK	64
	192..223	MethodId	32
	224..255	Kms2	32
	256..287	Kmp2	32
3 Reconnect, rekeying with ECDHE, new cryptosuite	0..63	MSK	64
	64..127	EMSK	64
	128..191	AMSK	64
	192..223	MethodId	32
	224..255	Kms2	32
	256..287	Kmp2	32
	288..319	Kz	32

Table 5: Key derivation output

Finally, every EAP method must export a Server-Id, Peer-Id and Session-Id [RFC5247]. In EAP-NOOB, the exported Peer-Id is the PeerId which the server has assigned to the peer. The exported Server-Id is a zero-length string (i.e., null string) because EAP-NOOB neither knows nor assigns any server identifier. The exported Session-Id is created by concatenating the Type-Code xxx (TBA) with the MethodId, which is obtained from the KDF output as shown in Table 5.

3.6. Error handling

Various error conditions in EAP-NOOB are handled by sending an error notification message (Type=0) instead of the expected next EAP request or response message. Both the EAP server and the peer may send the error notification, as shown in Figure 9 and Figure 10. After sending or receiving an error notification, the server MUST send an EAP-Failure (as required by [RFC3748] section 4.2). The

notification MAY contain an ErrorInfo field, which is a UTF-8 encoded text string with a maximum length of 500 bytes. It is used for sending descriptive information about the error for logging and debugging purposes.

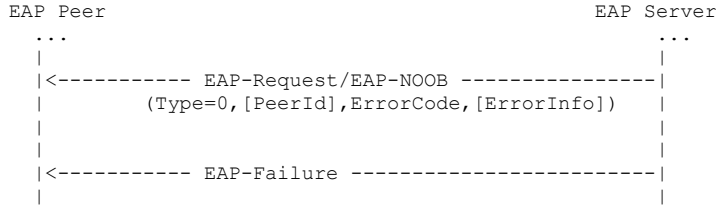

Figure 9: Error notification from server to peer

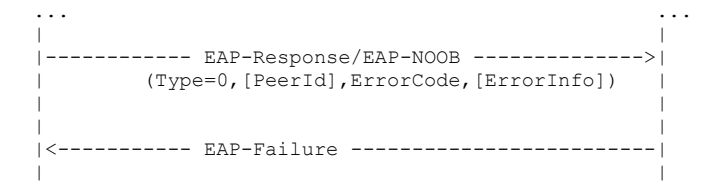

Figure 10: Error notification from peer to server

After the exchange fails due to an error notification, the server and peer set the association state as follows. In the Initial Exchange, both the sender and recipient of the error notification MUST set the association state to the Unregistered (0) state. In the Waiting and Completion Exchanges, each side MUST remain in its old state as if the failed exchange had not taken place, with the exception that the recipient of error code 2003 processes it as specified in Section 3.2.4. In the Reconnect Exchange, both sides MUST set the association state to the Reconnecting (3) state.

Errors that occur in the OOB channel are not explicitly notified in-band.

3.6.1. Invalid messages

If the NAI structure is invalid, the server SHOULD send the error code 1001 to the peer. The recipient of an EAP-NOOB request or response SHOULD send the following error codes back to the sender: 1002 if it cannot parse the message as a JSON object or the top-level JSON object has missing or unrecognized members; 1003 if a data field has an invalid value, such as an integer out of range, and there is no more specific error code available; 1004 if the received message type was unexpected in the current state; 2004 if the PeerId has an unexpected value; 2003 if the NoobId is not recognized; and 1007 if the ECDHE key is invalid.

3.6.2. Unwanted peer

The preferred way for the EAP server to rate limit EAP-NOOB connections from a peer is to use the SleepTime parameter in the Waiting Exchange. However, if the EAP server receives repeated EAP-NOOB connections from a peer which apparently should not connect to this server, the server MAY indicate that the connections are unwanted by sending the error code 2001. After receiving this error message, the peer MAY refrain from reconnecting to the same EAP server and, if possible, both the EAP server and peer SHOULD indicate this error condition to the user or server administrator. However, in order to avoid persistent denial of service, the peer is not required to stop entirely from reconnecting to the server.

3.6.3. State mismatch

In the states indicated by "-" in Figure 11 in Appendix A, user action is required to reset the association state or to recover it, for example, from backup storage. In those cases, the server sends the error code 2002 to the peer. If possible, both the EAP server and peer SHOULD indicate this error condition to the user or server administrator.

3.6.4. Negotiation failure

If there is no matching protocol version, the peer sends the error code 3001 to the server. If there is no matching cryptosuite, the peer sends the error code 3002 to the server. If there is no matching OOB direction, the peer sends the error code 3003 to the server.

In practice, there is no way of recovering from these errors without software or hardware changes. If possible, both the EAP server and peer SHOULD indicate these error conditions to the user.

3.6.5. Cryptographic verification failure

If the receiver of the OOB message detects an unrecognized PeerId or incorrect fingerprint (Hoob) in the OOB message, the receiver MUST remain in the Waiting for OOB state (1) as if no OOB message was received. The receiver SHOULD indicate the failure to accept the OOB message to the user. No in-band error message is sent.

Note that if the OOB message was delivered from the server to the peer and the peer does not recognize the PeerId, the likely cause is that the user has unintentionally delivered the OOB message to the wrong peer device. If possible, the peer SHOULD indicate this to the user; however, the peer device may not have the capability for many different error indications to the user and it MAY use the same indication as in the case of an incorrect fingerprint.

The rationale for the above is that the invalid OOB message could have been presented to the receiver by mistake or intentionally by a malicious party and, thus, it should be ignored in the hope that the honest user will soon deliver a correct OOB message.

If the EAP server or peer detects an incorrect message authentication code (MACs, MACp, MACs2, MACp2), it sends the error code 4001 to the other side. As specified in the beginning of Section 3.6, the failed Completion Exchange will not result in server or peer state changes while an error in the Reconnect Exchange will put both sides to the Reconnecting (3) state and thus lead to another reconnect attempt.

The rationale for this is that the invalid cryptographic message may have been spoofed by a malicious party and, thus, it should be ignored. In particular, a spoofed message on the in-band channel should not force the honest user to perform the OOB Step again. In practice, however, the error may be caused by other failures, such as a software bug. For this reason, the EAP server MAY limit the rate of peer connections with SleepTime after the above error. Also, there SHOULD be a way for the user to reset the peer to the Unregistered state (0), so that the OOB Step can be repeated as the last resort.

3.6.6. Application-specific failure

Applications MAY define new error messages for failures that are specific to the application or to one type of OOB channel. They MAY also use the generic application-specific error code 5001, or the error codes 5002 and 5004, which have been reserved for indicating invalid data in the ServerInfo and PeerInfo fields, respectively. Additionally, anticipating OOB channels that make use of a URL, the error code 5003 has been reserved for indicating invalid server URL.

4. IANA Considerations

This section provides guidance to the Internet Assigned Numbers Authority (IANA) regarding registration of values related to the EAP-NOOB protocol, in accordance with [RFC8126].

The EAP Method Type number for EAP-NOOB needs to be assigned.

This memo also requires IANA to create new registries as defined in the following subsections.

4.1. Cryptosuites

Cryptosuites are identified by an integer. Each cryptosuite MUST specify an ECDHE curve for the key exchange, encoding of the ECDHE public key as a JWK object, and a cryptographic hash function for the fingerprint and HMAC computation and key derivation. The hash value output by the cryptographic hash function MUST be at least 32 bytes in length. The following suites are defined by EAP-NOOB:

Cryptosuite	Algorithms
1	ECDHE curve Curve25519 [RFC7748], public-key format [RFC7517] Section 6.2.1, hash function SHA-256 [RFC6234]
2	ECDHE curve NIST P-256 [FIPS186-4], public-key format [RFC7517] Section 6.2.1, hash function SHA-256 [RFC6234]

Table 6: EAP-NOOB cryptosuites

EAP-NOOB implementations MUST support Cryptosuite 1. Support for Cryptosuite 2 is RECOMMENDED. An example of Cryptosuite 1 public-key encoded as a JWK object is given below (line breaks are for readability only).

"jwk":{"kty":"EC","crv":"Curve25519","x":"3p7bfXt9wbTTW2HC7OQ1Nz-DQ8hbeGdNrfx-FG-IK08"}

Assignment of new values for new cryptosuites MUST be done through IANA with "Specification Required" and "IESG Approval" as defined in [RFC8126].

4.2. Message Types

EAP-NOOB request and response pairs are identified by an integer Message Type. The following Message Types are defined by EAP-NOOB:

Message Type	Used in Exchange	Purpose
1	All exchanges	PeerId and PeerState discovery
2	Initial	Version, cryptosuite and parameter negotiation
3	Initial	Exchange of ECDHE keys and nonces
4	Waiting	Indication to peer that the server has not yet received an OOB message
5	Completion	NoobId discovery
6	Completion	Authentication and key confirmation with HMAC
7	Reconnect	Version, cryptosuite, and parameter negotiation
8	Reconnect	Exchange of ECDHE keys and nonces
9	Reconnect	Authentication and key confirmation with HMAC
0	Error	Error notification

Table 7: EAP-NOOB Message Types

Assignment of new values for new Message Types MUST be done through IANA with "Expert Review" as defined in [RFC8126].

4.3. Error codes

The error codes defined by EAP-NOOB are listed in Table 8.

Error code	Purpose
1001	Invalid NAI
1002	Invalid message structure
1003	Invalid data
1004	Unexpected message type
1007	Invalid ECDHE key
2001	Unwanted peer
2002	State mismatch, user action required
2003	Unrecognized OOB message identifier
2004	Unexpected peer identifier
3001	No mutually supported protocol version
3002	No mutually supported cryptosuite
3003	No mutually supported OOB direction
4001	HMAC verification failure
5001	Application-specific error
5002	Invalid server info
5003	Invalid server URL
5004	Invalid peer info
6001-6999	Private and experimental use

Table 8: EAP-NOOB error codes

Assignment of new error codes MUST be done through IANA with "Specification Required" and "IESG Approval" as defined in [RFC8126], with the exception of the range 6001-6999, which is reserved for "Private Use" and "Experimental Use".

Commented [DT26]: https://tools.ietf.org/html/rfc8126#section-4.2 says:
"When code points are set aside for Experimental Use, it's important to make clear any expected restrictions on experimental scope. For example, say whether it's acceptable to run experiments using those code points over the open Internet or whether such experiments should be confined to more closed environments. See [RFC6994] for an example of such considerations."

4.4. Domain name reservation considerations

A special-use domain (.arpa) should be registered to replace the temporary domain "eap-noob.net" used in this draft. The considerations required by [RFC6761] for registering this special-use domain name are the following:

Commented [DT27]: Suggest stating the registry explicitly: https://www.iana.org/domains/arpa

- Users: Non-admin users are not expected to encounter this name or recognize it as special. AAA administrators may need to recognize the name.
- Application Software: Application software is not expected to recognize this domain name as special.
- Name Resolution APIs and Libraries: Name resolution APIs and libraries are not expected to recognize this domain name as special.

- Caching DNS Servers: Caching servers are not expected to recognize this domain name as special.
- Authoritative DNS Servers: Authoritative DNS servers MUST respond to queries for eap-noob.net (.arpa domain TBA) with NXDOMAIN.
- DNS Server Operators: Except for the authoritative DNS server, there are no special requirements for the operators.
- DNS Registries/Registrars: There are no special requirements for DNS registrars.

5. Implementation Status

This section records the status of known implementations of the protocol defined by this specification at the time of posting of this Internet-Draft, and is based on a proposal described in [RFC7942]. The description of implementations in this section is intended to assist the IETF in its decision processes in progressing drafts to RFCs. Please note that the listing of any individual implementation here does not imply endorsement by the IETF. Furthermore, no effort has been spent to verify the information presented here that was supplied by IETF contributors. This is not intended as, and must not be construed to be, a catalog of available implementations or their features. Readers are advised to note that other implementations may exist.

5.1. Implementation with wpa_supplicant and hostapd

- Responsible Organization: Aalto University
- Location: <https://github.com/tuomaura/eap-noob>
- Coverage: This implementation includes all of the features described in the current specification. The implementation supports two dimensional QR codes and NFC as example out-of-band (OOB) channels.
- Level of Maturity: Alpha
- Version compatibility: Version 06 of the draft implemented
- Licensing: BSD
- Contact Information: Tuomas Aura, tuomas.aura@aalto.fi

5.2. Implementation on Contiki

- Responsible Organization: University of Murcia and Aalto University
- Location: <https://github.com/eduingles/coap-eap-noob>
- Coverage: This implementation includes all of the features described in the current specification. The implementation uses a blinking LED light as the out-of-band (OOB) channel.
- Level of Maturity: Alpha
- Version compatibility: Version 05 of the draft implemented
- Licensing: BSD
- Contact Information: Eduardo Ingles, eduardo.ingles@um.es

5.3. Protocol modeling

The current EAP-NOOB specification has been modeled with the mCRL2 formal specification language [mcrl2]. The model <https://github.com/tuomaura/eap-noob/tree/master/protocolmodel/mcrl2> was used mainly for simulating the protocol behavior and for verifying basic safety and liveness properties as part of the specification process. For example, we verified the correctness of the tiebreaking mechanism when two OOB messages are received simultaneously, one in each direction. We also verified that a man-in-the-middle attacker cannot cause persistent failure by spoofing a finite number of messages in the Reconnect Exchange. Additionally, the protocol has been modeled with the ProVerif [proverif] tool. This model <https://github.com/tuomaura/eap-noob/tree/master/protocolmodel/proverif> was used to verify security properties such as mutual authentication.

6. Security considerations

EAP-NOOB is an authentication and key derivation protocol and, thus, security considerations can be found in most sections of this specification. In the following, we explain the protocol design and highlight some other special considerations.

6.1. Authentication principle

EAP-NOOB establishes a shared secret with an authenticated ECDHE key exchange. The mutual authentication in EAP-NOOB is based on two separate features, both conveyed in the OOB message. The first

authentication feature is the secret nonce Noob. The peer and server use this secret in the Completion Exchange to mutually authenticate the session key previously created with ECDHE. The message authentication codes computed with the secret nonce Noob are alone sufficient for authenticating the key exchange. The second authentication feature is the integrity-protecting fingerprint Hoob. Its purpose is to prevent impersonation and man-in-the-middle attacks even in situations where the attacker is able to eavesdrop on the OOB channel and the nonce Noob is compromised. In some human-assisted OOB channels, such as human-perceptible audio or a user-typed URL, it may be easier to detect tampering than ~~spying~~ disclosure of the OOB message, and such applications benefit from the second authentication feature.

Commented [DT28]: Suggested grammar fix ("spying of" sounds wrong to me, usually one would say "spying on")

The additional security provided by the cryptographic fingerprint Hoob is somewhat intricate to understand. The endpoint that receives the OOB message uses Hoob to verify the integrity of the ECDHE exchange. Thus, the OOB receiver can detect impersonation and man-in-the-middle attacks that may have happened on the in-band channel. The other endpoint, however, is not equally protected because the OOB message and fingerprint are sent only in one direction. Some protection to the OOB sender is afforded by the fact that the user may notice the failure of the association at the OOB receiver and therefore reset the OOB sender. Other device-pairing protocols have solved similar situations by requiring the user to confirm to the OOB sender that the association was accepted by the OOB receiver, e.g., with a button press on the sender side. Applications MAY implement EAP-NOOB in this way. Nevertheless, since EAP-NOOB was designed to work with strictly one-directional OOB communication and the fingerprint is only the second authentication feature, the EAP-NOOB specification does not mandate such explicit confirmation to the OOB sender.

To summarize, EAP-NOOB uses the combined protection of the secret nonce Noob and the cryptographic fingerprint Hoob, both conveyed in the OOB message. The secret nonce Noob alone is sufficient for mutual authentication, unless the attacker can eavesdrop on it from the OOB channel. Even if an attacker is able to eavesdrop on the secret nonce Noob, it nevertheless cannot perform a full man-in-the-middle attack on the in-band channel because a mismatching fingerprint would alert the OOB receiver, which would reject the OOB message. The attacker that eavesdropped on the secret nonce can impersonate the OOB receiver to the OOB sender. If it does, the association will appear to be complete only on the OOB sender side, and such situations have to be resolved by the user by resetting the OOB sender to the initial state.

The expected use cases for EAP-NOOB are ones where it replaces a user-entered access credential in IoT appliances. In wireless

network access without EAP, the user-entered credential is often a passphrase that is shared by all the network stations. The advantage of an EAP-based solution, including EAP-NOOB, is that it establishes a different master secret for each peer device, which makes the system more resilient against device compromise. Another advantage is that it is possible to revoke the security association for an individual device on the server side.

Forward secrecy in EAP-NOOB is optional. The Reconnect Exchange in EAP-NOOB provides forward secrecy only if both the server and peer send their fresh ECDHE keys. This allows both the server and the peer to limit the frequency of the costly computation that is required for forward secrecy. The server MAY adjust the frequency of its attempts at ECDHE rekeying based on what it knows about the peer's computational capabilities.

The users delivering the OOB messages will often authenticate themselves to the EAP server, e.g., by logging into a secure web page or API. In this case, the server can reliably associate the peer device with the user account. Applications that make use of EAP-NOOB can use this information for configuring the initial owner of the freshly-registered device.

6.2. Identifying correct endpoints

Potential weaknesses in EAP-NOOB arise from the fact that the user must identify physically the correct peer device. If the attacker is able to trick the user into delivering the OOB message to or from the wrong peer device, the server may create an association with the wrong peer. This reliance on the user in identifying the correct endpoints is an inherent property of user-assisted out-of-band authentication.

It is, however, not possible to exploit accidental delivery of the OOB message to the wrong device when the user makes an unexpected mistake. This is because the wrong peer device would not have prepared for the attack by performing the Initial Exchange with the server. In comparison, simpler security bootstrapping solutions where the master key is transferred to the device via the OOB channel are vulnerable to opportunistic attacks if the user mistakenly delivers the master key to the wrong device or to more than one device.

One mechanism that can mitigate user mistakes is certification of peer devices. The certificate can convey to the server authentic identifiers and attributes, such as model and serial number, of the peer device. Compared to a fully certificate-based authentication, however, EAP-NOOB can be used without trusted third parties and does

not require the user to know any identifier of the peer device; physical access to the device is sufficient for bootstrapping with EAP-NOOB.

Similarly, the attacker can try to trick the user into delivering the OOB message to the wrong server, so that the peer device becomes associated with the wrong server. If the EAP server is accessed through a web user interface, the attack is akin to phishing attacks where the user is tricked into accessing the wrong URL and wrong web page. OOB implementation with a dedicated app on a mobile device, which communicates with a server API at a pre-configured URL, can protect against such attacks.

6.3. Trusted path issues and misbinding attacks

Another potential threat is spoofed user input or output on the peer device. When the user is delivering the OOB message to or from the correct peer device, a trusted path between the user and the peer device is needed. That is, the user must communicate directly with an authentic operating system and EAP-NOOB implementation in the peer device and not with a spoofed user interface. Otherwise, a Registered device that is under the control of the attacker could emulate the behavior of an unregistered device. The secure path can be implemented, for example, by having the user press a reset button to return the device to the Unregistered state and to invoke a trusted UI. The problem with such trusted paths is that they are not standardized across devices.

Another potential consequence of a spoofed UI is the misbinding attack where the user tries to register a correct but compromised device, which tricks the user into registering another (uncompromised) device instead. For example, the compromised device might have a malicious full-screen app running, which presents to the user QR codes copied, in real time, from another device's screen. If the unwitting user scans the QR code and delivers the OOB message in it to the server, the wrong device may become registered in the server. Such misbinding vulnerabilities arise because the user does not have any secure way of verifying that the in-band cryptographic handshake and the out-of-band physical access are terminated at the same physical device. Sethi et al. [Sethi19] analyze the misbinding threat against device-pairing protocols and also EAP-NOOB. Essentially, all protocols where the authentication relies on the user's physical access to the device are vulnerable to misbinding, including EAP-NOOB.

A standardized trusted path for communicating directly with the trusted computing base in a physical device would mitigate the misbinding threat, but such paths rarely exist in practice. Careful

Commented [DT29]: Why is this capitalized?

Commented [DT30]: Another attack is where a rogue device keeps generating random IDs (MAC, etc.) in order to cause havoc with the server, whether that's a memory attack, or an attempt to cause so much UI noise to users that it DoS's the ability to add valid devices.

asset tracking on the server side can also prevent most misbinding attacks if the peer device sends its identifiers or attributes in the PeerInfo field and the server compares them with the expected values. The wrong but uncompromised device's PeerInfo will not match the expected values. Device certification by the manufacturer can further strengthen the asset tracking.

6.4. Peer identifiers and attributes

The PeerId value in the protocol is a server-allocated identifier for its association with the peer and SHOULD NOT be shown to the user because its value is initially ephemeral. Since the PeerId is allocated by the server and the scope of the identifier is the single server, the so-called identifier squatting attacks, where a malicious peer could reserve another peer's identifier, are not possible in EAP-NOOB. The server SHOULD assign a random or pseudo-random PeerId to each new peer. It SHOULD NOT select the PeerId based on any peer characteristics that it may know, such as the peer's link-layer network address.

User reset or failure in the OOB Step can cause the peer to perform many Initial Exchanges with the server, to allocate many PeerId values, and to store the ephemeral protocol state for them. The peer will typically only remember the latest one. EAP-NOOB leaves it to the implementation to decide when to delete these ephemeral associations. There is no security reason to delete them early, and the server does not have any way to verify that the peers are actually the same one. Thus, it is safest to store the ephemeral states for at least one day. If the OOB messages are sent only in the server-to-peer direction, the server SHOULD NOT delete the ephemeral state before all the related Noob values have expired.

After completion of EAP-NOOB, the server may store the PeerInfo data, and the user may use it to identify the peer and its attributes, such as the make and model or serial number. A compromised peer could lie in the PeerInfo that it sends to the server. If the server stores any information about the peer, it is important that this information is approved by the user during or after the OOB Step. Without verification by the user or authentication with vendor certificates on the application level, the PeerInfo is not authenticated information and should not be relied on.

Commented [DT31]: The PeerInfo can be anything, right? So draft-ietf-rats-eat could be used to provide attestability of the information for example.

One possible use for the PeerInfo field is EAP channel binding ([RFC3748] Section 7.15). That is, the peer MAY include in PeerInfo any data items that it wants to bind to the ~~the~~ EAP-NOOB association and to the exported keys. These can be properties of the authenticator or the access link, such as the SSID and BSSID of the wireless network (see Appendix C).

6.5. Identity protection

The PeerInfo field contains identifiers and other information about the peer device (see Appendix C), and the peer sends this information in plaintext to the EAP server before the server authentication in EAP-NOOB has been completed. While the information refers to the peer device and not directly to the user, it may be better for user privacy to avoid sending unnecessary information. In the Reconnect Exchange, the optional PeerInfo SHOULD be omitted unless some critical data has changed.

Peer devices that randomize their layer-2 address to prevent tracking can do this whenever the user resets the EAP-NOOB association. During the lifetime of the association, the PeerId is a unique identifier that can be used to track the peer in the access network. Later versions of this specification may consider updating the PeerId at each Reconnect Exchange. In that case, it is necessary to consider how the authenticator and access-network administrators can recognize and blacklist misbehaving peer devices and how to avoid loss of synchronization between the server and the peer if messages are lost during the identifier update.

6.6. Downgrading threats

The fingerprint Hoob protects all the information exchanged in the Initial Exchange, including the cryptosuite negotiation. The message authentication codes MACs and MACp also protect the same information. The message authentication codes MACs2 and MACp2 protect information exchanged during key renegotiation in the Reconnect Exchange. This prevents downgrading attacks to weaker cryptosuites as long as the possible attacks take more time than the maximum time allowed for the EAP-NOOB completion. This is typically the case for recently discovered cryptanalytic attacks.

As an additional precaution, the EAP server and peer MUST check for downgrading attacks in the Reconnect Exchange as follows. As long as the server or peer saves any information about the other endpoint, it MUST also remember the previously negotiated cryptosuite and MUST NOT accept renegotiation of any cryptosuite that is known to be weaker than the previous one, such as a deprecated cryptosuite.

Integrity of the direction negotiation cannot be verified in the same way as the integrity of the cryptosuite negotiation. That is, if the OOB channel used in an application is critically insecure in one direction, a man-in-the-middle attacker could modify the negotiation messages and thereby cause that direction to be used. Applications that support OOB messages in both directions SHOULD therefore ensure that the OOB channel has sufficiently strong security in both

directions. While this is a theoretical vulnerability, it could arise in practice if EAP-NOOB is deployed in new applications. Currently, we expect most peer devices to support only one OOB direction, in which case interfering with the direction negotiation can only prevent the completion of the protocol.

The long-term shared key material Kz in the persistent EAP-NOOB association is established with an ECDHE key exchange when the peer and server are first associated. It is a weaker secret than a manually configured random shared key because advances in cryptanalysis against the used ECDHE curve could eventually enable the attacker to recover Kz. EAP-NOOB protects against such attacks by allowing cryptosuite upgrades in the Reconnect Exchange and by updating the shared key material Kz whenever the cryptosuite is upgraded. We do not expect the cryptosuite upgrades to be frequent, but if an upgrade becomes necessary, it can be done without manual reset and reassociation of the peer devices.

6.7. Recovery from loss of last message

The EAP-NOOB Completion Exchange, as well as the Reconnect Exchange with cryptosuite update, result in a persistent state change that should take place either on both endpoints or on neither; otherwise, the result is a state mismatch that requires user action to resolve. The state mismatch can occur if the final EAP response of the exchanges is lost. In the Completion Exchange, the loss of the final response (Type=6) results in the peer moving to Registered (4) state and creating a persistent EAP-NOOB association while the server stays in an ephemeral state (1 or 2). In the Reconnect Exchange, the loss of the final response (Type=9) results in the peer moving to the Registered (4) state and updating its persistent key material Kz while the server stays in the Reconnecting (3) state and keeps the old key material.

The state mismatch is an example of an unavoidable problem in distributed systems: it is theoretically impossible to guarantee synchronous state changes in endpoints that communicate asynchronously. The protocol will always have one critical message that may get lost, so that one side commits to the state change and the other side does not. In EAP, the critical message is the final response from the peer to the server. While the final response is normally followed by EAP-Success, [RFC3748] section 4.2 states that the peer MAY assume that the EAP-Success was lost and the authentication was successful. Furthermore, EAP method implementations in the peer do not receive notification of the EAP-Success message from the parent EAP state machine [RFC4137]. For these reasons, EAP-NOOB on the peer side commits to a state change already when it sends the final response.

The best available solution to the loss of the critical message is to keep trying. EAP retransmission behavior defined in Section 4.3 of [RFC3748] suggests 3-5 retransmissions. In the absence of an attacker, this would be sufficient to reduce the probability of failure to an acceptable level. However, a determined attacker on the in-band channel can drop the final EAP-Response message and all subsequent retransmissions. In the Completion Exchange (KeyingMode=0) and in the Reconnect Exchange with cryptosuite upgrade (KeyingMode=3), this could result in a state mismatch and persistent denial of service until user resets the peer state.

EAP-NOOB implements its own recovery mechanism that allows unlimited retries of the Reconnect Exchange. When the DoS attacker eventually stops dropping packets on the in-band channel, the protocol will recover. The logic for this recovery mechanism is specified in Section 3.4.2.

EAP-NOOB does not implement the same kind of retry mechanism in the Completion Exchange. The reason is that there is always a user involved in the initial association process, and the user can repeat the OOB Step to complete the association after the DoS attacker has left. On the other hand, Reconnect Exchange needs to work without user involvement.

6.8. EAP security claims

EAP security claims are defined in section 7.2.1 of [RFC3748]. The security claims for EAP-NOOB are listed in Table 9.

Security property	EAP-NOOB claim
Authentication mechanism	ECDHE key exchange with out-of-band authentication
Protected cryptosuite negotiation	yes
Mutual authentication	yes
Integrity protection	yes
Replay protection	yes
Key derivation	yes
Key strength	The specified cryptosuites provide key strength of at least 128 bits.
Dictionary attack protection	yes
Fast reconnect	yes
Cryptographic binding	not applicable
Session independence	yes
Fragmentation	no
Channel binding	yes (The ServerInfo and PeerInfo can be used to convey integrity-protected channel properties such as network SSID or peer MAC address.)

Table 9: EAP security claims

Commented [DT32]: RFC3748 section 7.2.1 says "This list of security claims is not exhaustive. **Additional properties, such as additional denial-of-service protection, may be relevant as well**."

Per section 6.5., I think there should be a row added: Identity Protection: No

7. References

7.1. Normative references

[FIPS186-4] National Institute of Standards and Technology, "Digital Signature Standard (DSS)", FIPS 186-4 , July 2013.

[NIST-DH] Barker, E., Chen, L., Roginsky, A., Vassilev, A., and R. Davis, "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography", NIST Special Publication 800-56A Revision 3 , April 2018, <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar3.pdf>.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/info/rfc2104>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3748] Aboba, B., Blunk, L., Vollbrecht, J., Carlson, J., and H. Levkowetz, Ed., "Extensible Authentication Protocol (EAP)", RFC 3748, DOI 10.17487/RFC3748, June 2004, <https://www.rfc-editor.org/info/rfc3748>.

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

[RFC5247] Aboba, B., Simon, D., and P. Eronen, "Extensible Authentication Protocol (EAP) Key Management Framework", RFC 5247, DOI 10.17487/RFC5247, August 2008, <https://www.rfc-editor.org/info/rfc5247>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/info/rfc6234>.

[RFC6761] Cheshire, S. and M. Krochmal, "Special-Use Domain Names", RFC 6761, DOI 10.17487/RFC6761, February 2013, <https://www.rfc-editor.org/info/rfc6761>.

[RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <https://www.rfc-editor.org/info/rfc7517>.

[RFC7542] DeKok, A., "The Network Access Identifier", RFC 7542, DOI 10.17487/RFC7542, May 2015, <https://www.rfc-editor.org/info/rfc7542>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/info/rfc7748>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

7.2. Informative references

[BluetoothPairing] Bluetooth, SIG, "Simple pairing whitepaper", Technical report , 2007.

[EUI-48] Institute of Electrical and Electronics Engineers, "802-2014 IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture.", IEEE Standard 802-2014. , June 2014.

[IEEE-802.1X] Institute of Electrical and Electronics Engineers, "Local and Metropolitan Area Networks: Port-Based Network Access Control", IEEE Standard 802.1X-2004. , December 2004.

[mcrl2] Groote, J. and M. Mousavi, "Modeling and analysis of communicating systems", The MIT press , 2014, <https://mitpress.mit.edu/books/modeling-and-analysis-communicating-systems>.

[proverif]
Blanchet, B., Smyth, B., Cheval, V., and M. Sylvestre, "ProVerif 2.00: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial", The MIT press , 2018, <http://prosecco.gforge.inria.fr/personal/bblanche/proverif/>.

[RFC2904] Vollbrecht, J., Calhoun, P., Farrell, S., Gommans, L., Gross, G., de Bruijn, B., de Laat, C., Holdrege, M., and D. Spence, "AAA Authorization Framework", RFC 2904, DOI 10.17487/RFC2904, August 2000, <https://www.rfc-editor.org/info/rfc2904>.

[RFC4137] Vollbrecht, J., Eronen, P., Petroni, N., and Y. Ohba, "State Machines for Extensible Authentication Protocol (EAP) Peer and Authenticator", RFC 4137, DOI 10.17487/RFC4137, August 2005, <https://www.rfc-editor.org/info/rfc4137>.

[RFC4266] Hoffman, P., "The gopher URI Scheme", RFC 4266, DOI 10.17487/RFC4266, November 2005, <https://www.rfc-editor.org/info/rfc4266>.

[RFC5216] Simon, D., Aboba, B., and R. Hurst, "The EAP-TLS Authentication Protocol", RFC 5216, DOI 10.17487/RFC5216, March 2008, <https://www.rfc-editor.org/info/rfc5216>.

[RFC7942] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", BCP 205, RFC 7942, DOI 10.17487/RFC7942, July 2016, <https://www.rfc-editor.org/info/rfc7942>.

[Sethi14] Sethi, M., Oat, E., Di Francesco, M., and T. Aura, "Secure Bootstrapping of Cloud-Managed Ubiquitous Displays", Proceedings of ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp 2014), pp. 739-750, Seattle, USA , September 2014, <http://dx.doi.org/10.1145/2632048.2632049>.

[Sethi19] Sethi, M., Peltonen, A., and T. Aura, "Misbinding Attacks on Secure Device Pairing", 2019, <https://arxiv.org/abs/1902.07550>.

Appendix A. Exchanges and events per state

Figure 11 shows how the EAP server chooses the exchange type depending on the server and peer states. In the state combinations marked with hyphen "-", there is no possible exchange and user action is required to make progress. Note that peer state 4 is omitted from the table because the peer never connects to the server when the peer is in that state. The table also shows the handling of errors in each exchange. A notable detail is that the recipient of error code 2003 moves to state 1.

peer states	exchange chosen by server	next peer and server states
server state: Unregistered (0)		
0..2	Initial Exchange	both 1 (0 on error)
3	-	no change, notify user
server state: Waiting for OOB (1)		
0	Initial Exchange	both 1 (0 on error)
1	Waiting Exchange	both 1 (no change on error)
2	Completion Exchange	both 4 (A)
3	-	no change, notify user
server state: OOB Received (2)		
0	Initial Exchange	both 1 (0 on error)
1	Completion Exchange	both 4 (B)
2	Completion Exchange	both 4 (A)
3	-	no change, notify user
server state: Reconnecting (3) or Registered (4)		
0..2	-	no change, notify user
3	Reconnect Exchange	both 4 (3 on error)

(A) peer to 1 on error 2003, no other changes on error
(B) server to 1 on error 2003, no other changes on error

Figure 11: How server chooses the exchange type

Figure 12 lists the local events that can take place in the server or peer. Both the server and peer output and accept OOB messages in

association state 1, leading the receiver to state 2. Communication errors and timeouts in states 0..2 lead back to state 0, while similar errors in states 3..4 lead to state 3. Application request for rekeying (e.g. to refresh session keys or to upgrade cryptosuite) also takes the association from state 3..4 to state 3. User can always reset the association state to 0. Recovering association data, e.g. from a backup, leads to state 3.

server/ peer state	possible local events on server and peer	next state
1	OOB Output*	1
1	OOB Input*	2 (1 on error)
0..2	Timeout/network failure	0
3..4	Timeout/network failure	3
3..4	Rekeying request	3
0..4	User resets peer state	0
0..4	Association state recovery	3

Figure 12: Local events on server and peer

Appendix B. Application-specific parameters

Table 10 lists OOB channel parameters that need to be specified in each application that makes use of EAP-NOOB. The list is not exhaustive and is included for the convenience of implementers only.

Parameter	Description
OobDirs	Allowed directions of the OOB channel
OobMessageEncoding	How the OOB message data fields are encoded for the OOB channel
SleepTimeDefault	Default minimum time in seconds that the peer should sleep before the next Waiting Exchange
OobRetries	Number of received OOB messages with invalid Hoob after which the receiver moves to Unregistered (0) state
NoobTimeout	How many seconds the sender of the OOB message remembers the sent Noob value. The RECOMMENDED value is 3600 seconds.
ServerInfoMembers	Required members in ServerInfo
PeerInfoMembers	Required members in PeerInfo

Table 10: OOB channel characteristics

Appendix C. ServerInfo and PeerInfo contents

The ServerInfo and PeerInfo fields in the Initial Exchange and Reconnect Exchange enable the server and peer, respectively, to send information about themselves to the other endpoint. They contain JSON objects whose structure may be specified separately for each application and each type of OOB channel. ServerInfo and PeerInfo MAY contain auxiliary data needed for the OOB channel messaging and for EAP channel binding. Table 11 lists some suggested data fields for ServerInfo.

Commented [DT33]: It doesn't look like you're specifying an IANA registry to contain field names. Given that, how do you get interoperability? If every app and type of OOB channel specifies a different structure, how do you do capability negotiation to agree on which structure definition is being used? (Since two specifications might use the same field names for very different purposes, in theory.) I didn't see any identifier that can be used as the "type" of the ServerInfo or PeerInfo data.

Data field	Description
ServerName	String that may be used to aid human identification of the server.
ServerURL	Prefix string when the OOB message is formatted as a URL, as suggested in Appendix E.
SSIDList	List of wireless network identifier (SSID) strings used for roaming support, as suggested in Appendix D. JSON array of UTF-8 encoded SSID strings.
Base64SSIDList	List of wireless network identifier (SSID) strings used for roaming support, as suggested in Appendix D. JSON array of SSIDs, each of which is base64url encoded without padding. Peers SHOULD send at most one of the fields SSIDList and Base64SSIDList in PeerInfo, and the server SHOULD ignore SSIDList if Base64SSIDList is included.

Table 11: Suggested ServerInfo data fields

PeerInfo typically contains auxiliary information for identifying and managing peers on the application level at the server end. Table 12 lists some suggested data fields for PeerInfo.

Commented [DT34]: I think you mean “IEEE 802.11” or “Wi-Fi”, yes? Other network types (802.15.4, Bluetooth etc) are also “wireless” and would presumably need fields other than “SSIDList”. Is table 11 specific to 802.11? Or is this suggesting that SSIDList is a field not specific to 802.11 SSIDs?

Commented [DT35]: This is still pretty vague. Is it supposed to be normalized in any way (normalization form, half-width vs full-width, etc.)?

Data field	Description
PeerName	String that may be used to aid human identification of the peer.
Manufacturer	Manufacturer or brand string.
Model	Manufacturer-specified model string.
SerialNumber	Manufacturer-assigned serial number.
MACAddress	Peer link-layer identifier (EUI-48) in the 12-digit base-16 form [EUI-48]. The string MAY include additional colon ':' or dash '-' characters that MUST be ignored by the server.
SSID	Wireless network SSID for channel binding. The SSID is a UTF-8 string.
Base64SSID	Wireless network SSID for channel binding. The SSID is base64url encoded. Peer SHOULD send at most one of the fields SSID and Base64SSID in PeerInfo, and the server SHOULD ignore SSID if Base64SSID is included.
BSSID	Wireless network BSSID (EUI-48) in the 12-digit base-16 form [EUI-48]. The string MAY include additional colon ':' or dash '-' characters that MUST be ignored by the server.

Table 12: Suggested PeerInfo data fields

Commented [DT36]: And both upper case A-F and lower case a-f are allowed?

Commented [DT37]: Does normalization matter? Or can it be a different set of bytes that what the server puts in ServerInfo, for the same SSID?

Commented [DT38]: Same question about whether case matters

Appendix D. EAP-NOOB roaming

AAA architectures [RFC2904] allow for roaming of network-connected appliances that are authenticated over EAP. While the peer is roaming in a visited network, authentication still takes place between the peer and an authentication server at its home network. EAP-NOOB supports such roaming by assigning a Realm to the peer. After the Realm has been assigned, the peer's NAI enables the visited network to route the EAP session to the peer's home AAA server.

A peer device that is new or has gone through a hard reset should be connected first to the home network and establish an EAP-NOOB association with its home AAA server before it is able to roam.

After that, it can perform the Reconnect Exchange from the visited network.

Alternatively, the device may provide some method for the user to configure the Realm of the home network. In that case, the EAP-NOOB association can be created while roaming. The device will use the user-assigned Realm in the Initial Exchange, which enables the EAP messages to be routed correctly to the home AAA server.

While roaming, the device needs to identify the networks where the EAP-NOOB association can be used to gain network access. For 802.11 access networks, the server MAY send a list of SSID strings in the ServerInfo JSON object in a member called either SSIDList or Base64SSIDList. The list is formatted as explained in Table 11. If present, the peer MAY use this list as a hint to determine the networks where the EAP-NOOB association can be used for access authorization, in addition to the access network where the Initial Exchange took place.

Commented [DT39]: Is the peer responsible for normalizing the SSIDList Unicode strings if it needs to match the networks? Or is the server responsible for sending them in a normalized form that the AP expects?

Appendix E. OOB message as URL

While EAP-NOOB does not mandate any particular OOB communication channel, typical OOB channels include graphical displays and emulated NFC tags. In the peer-to-server direction, it may be convenient to encode the OOB message as a URL, which is then encoded as a QR code for displays and printers or as an NDEF record for NFC tags. A user can then simply scan the QR code or NFC tag and open the URL, which causes the OOB message to be delivered to the authentication server. The URL MUST specify the https protocol i.e., secure connection to the server, so that the man-in-the-middle attacker cannot read or modify the OOB message.

The ServerInfo in this case includes a JSON member called ServerUrl of the following format with maximum length of 60 characters:

Commented [DT40]: Just an observation: This limits applicability to servers that can get short hostnames (and paths).

```
https://<host>[:<port>]/[<path>]
```

To this, the peer appends the OOB message fields (PeerId, Noob, Hoob) as a query string. PeerId is provided to the peer by the server and might be a 22-character string. The peer base64url encodes, without padding, the 16-byte values Noob and Hoob into 22-character strings. The query parameters MAY be in any order. The resulting URL is of the following format:

```
https://<host>[:<port>]/[<path>]?P=<PeerId>&N=<Noob>&H=<Hoob>
```

The following is an example of a well-formed URL encoding the OOB message (without line breaks):

https://example.com/Noob?P=ZrD7qkczNoHGbGcN2bN0&N=rMinS0-F4EfCU8D9ljx
X_A&H=QvnMp4UGxuQVFaXPW_14UW

Appendix F. Example messages

The message examples in this section are generated with Curve25519 ECDHE test vectors specified in section 6.1 of [RFC7748] (server=Alice, peer=Bob). The direction of the OOB channel negotiated is 2 (server-to-peer). The JSON messages are as follows (line breaks are for readability only).

====== Initial Exchange ======

Identity response:
noob@eap-noob.net

EAP request (type 1):
{"Type":1}

EAP response (type 1):
{"Type":1,"PeerState":0}

EAP request (type 2):
{"Type":2,"Vers":[1],"PeerId":"07KRU6OgqX0HIeRFldnbSW","Realm":"no
ob.example.com","Cryptosuites":[1,2],"Dirs":3,"ServerInfo":{"Name"
:"Example","Url":"https://noob.example.com/sendOOB"}}

EAP response (type 2):
{"Type":2,"Verp":1,"PeerId":"07KRU6OgqX0HIeRFldnbSW","Cryptosuitep
":1,"Dirp":2,"PeerInfo":{"Make":"Acme","Type":"None","Serial":"DU-
9999","SSID":"Noob1","BSSID":"6c:19:8f:83:c2:80"}}

EAP request (type 3):
{"Type":3,"PeerId":"07KRU6OgqX0HIeRFldnbSW","PKs":{"kty":"EC","crv
":"Curve25519","x":"hSDwCYkwp1R0i33ctD73Wg2_Og0mOBr066SpjqqbTmo"},
"Ns":"PYO7NVd9Af3BxEri1MI6hL8Ck49YxwCjSRPqlC1SPbw","SleepTime":60}

EAP response (type 3):
{"Type":3,"PeerId":"07KRU6OgqX0HIeRFldnbSW","PKp":{"kty":"EC","crv
":"Curve25519","x":"3p7bfXt9wbTTW2HC7OQ1Nz-DQ8hbeGdNrfx-FG-
IK08"},"Np":"HIvB6g0n2btpxEcU7YXnWB-451ED6L6veQQd6ugiPFU"}

====== Waiting Exchange ======

Identity response:
noob@eap-noob.net

EAP request (type 1):

```
      {"Type":1}

   EAP response (type 1):
      {"Type":1,"PeerId":"07KRU6OgqX0HIeRFldnbSW","PeerState":1}

   EAP request (type 4):
      {"Type":4,"PeerId":"07KRU6OgqX0HIeRFldnbSW","SleepTime":60}

   EAP response (type 4):
      {"Type":4,"PeerId":"07KRU6OgqX0HIeRFldnbSW"}

   ====== OOB Step ======

   OOB message:
      P=07KRU6OgqX0HIeRFldnbSW&N=x3JlolaPciK4Wa6XlMJxtQ&H=65JT5zEgQm6UvI
      8ySRKoDA

   ====== Completion Exchange ======

   Identity response:
      noob@eap-noob.net

   EAP request (type 1):
      {"Type":1}

   EAP response (type 1):
      {"Type":1,"PeerId":"07KRU6OgqX0HIeRFldnbSW","PeerState":2}

   EAP request (type 5):
      {"Type":5,"PeerId":"07KRU6OgqX0HIeRFldnbSW"}

   EAP response (type 5):
      {"Type":5,"PeerId":"07KRU6OgqX0HIeRFldnbSW","NoobId":"U0OHwYGCS4nE
      kzk2TPIE6g"}

   EAP request (type 6):
      {"Type":6,"PeerId":"07KRU6OgqX0HIeRFldnbSW","NoobId":"U0OHwYGCS4nE
      kzk2TPIE6g","MACs":"-RXjPiEONm7QzVQFFbz1dWPBJ4Yi6J6RxFf48VHp37I"}

   EAP response (type 6):
      {"Type":6,"PeerId":"07KRU6OgqX0HIeRFldnbSW","MACp":"Sug_rhASwVkfSV
      ENvu--j4IebX5qa6Sp3OkNWIqQ-kM"}

   ====== Reconnect Exchange ======

   Identity response:
      noob@eap-noob.net
```

EAP request (type 1):

```
{"Type":1}
```

EAP response (type 1):

```
{"Type":1,"PeerId":"07KRU6OgqX0HIeRFldnbSW","PeerState":3}
```

EAP request (type 7):

```
{"Type":7,"Vers":[1],"PeerId":"07KRU6OgqX0HIeRFldnbSW","Cryptosuit
es":[1,2],"Realm":"noob.example.com","ServerInfo":{"Name":"Example
","Url":"https://noob.example.com/sendOOB"}}
```

EAP response (type 7):

```
{"Type":7,"Verp":1,"PeerId":"07KRU6OgqX0HIeRFldnbSW","Cryptosuitep
":1,"PeerInfo":{"Make":"Acme","Type":"None","Serial":"DU-
9999","SSID":"Noob1","BSSID":"6c:19:8f:83:c2:80"}}
```

EAP request (type 8):

```
{"Type":8,"PeerId":"07KRU6OgqX0HIeRFldnbSW","KeyingMode":2,"Ns2":"
RDLahHBlIgnmL_F_xcynrHurLPkCsrp3G3B_S82WUF4"}
```

EAP response (type 8):

```
{"Type":8,"PeerId":"07KRU6OgqX0HIeRFldnbSW","Np2":"jN0_V4P0JoTqwI9
VHHQKd9ozUh7tQdc9ABd-j6oTy_4"}
```

EAP request (type 9):

```
{"Type":9,"PeerId":"07KRU6OgqX0HIeRFldnbSW","MACs2":"YQLFAdqKS7wRB
NQ_2rijGWUqR83i7pcLrYc9I_LizR0"}
```

EAP response (type 9):

```
{"Type":9,"PeerId":"07KRU6OgqX0HIeRFldnbSW","MACp2":"16gc-
WOtyDnrB-RvBIpVkfFpv2qjJwIJwTSwFeHF2BQ"}
```

Appendix G. TODO list

- Update EAP Method Type number assigned by IANA
- Update reserved domain name.

Appendix H. Version history

- Version 01:
  - Fixed Reconnection Exchange.
  - URL examples.
  - Message examples.

* Improved state transition (event) tables.

- Version 02:

  * Reworked the rekeying and key derivation.

  * Increased internal key lengths and in-band nonce and HMAC lengths to 32 bytes.

  * Less data in the persistent EAP-NOOB association.

  * Updated reference [NIST-DH] to Revision 2 (2013).

  * Shorter suggested PeerId format.

  * Optimized the example of encoding OOB message as URL.

  * NoobId in Completion Exchange to differentiate between multiple valid Noob values.

  * List of application-specific parameters in appendix.

  * Clarified the equivalence of Unregistered state and no state.

  * Peer SHOULD probe the server regardless of the OOB channel direction.

  * Added new error messages.

  * Realm is part of the persistent association and can be updated.

  * Clarified error handling.

  * Updated message examples.

  * Explained roaming in appendix.

  * More accurate definition of timeout for the Noob nonce.

  * Additions to security considerations.

- Version 03:

  * Clarified reasons for going to Reconnecting state.

  * Included Verp in persistent state.

  * Added appendix on suggested ServerInfo and PeerInfo fields.

    * Exporting PeerId and SessionId.

    * Explicitly specified next state after OOB Step.

    * Clarified the processing of an expired OOB message and unrecognized NoobId.

    * Enabled protocol version upgrade in Reconnect Exchange.

    * Explained handling of redundant received OOB messages.

    * Clarified where raw and base64url encoded values are used.

    * Cryptosuite must specify the detailed format of the JWK object.

    * Base64url encoding in JSON strings is done without padding.

    * Simplified explanation of PeerId, Realm and NAI.

    * Added error codes for private and experimental use.

    * Updated the security considerations.

  o Version 04:

    * Recovery from synchronization failure due to lost last response.

  o Version 05:

    * Kz identifier added to help recovery from lost last messages.

    * Error message codes changed for better structure.

    * Improved security considerations section.

  o Version 06:

    * Kz identifier removed to enable PeerId anonymization in the future.

    * Clarified text on when to use server-assigned realm.

    * Send PeerId and PeerState in a separate request-reponse pair, not in NAI.

    * New subsection for the common handshake in all exchanges to avoid repetition.

- Version 07:
  - Updated example messages.
  - Added pointers to new implementation in Contiki.
- Version 08:
  - Editorial improvements and corrections.
- WG Version 00:
  - Editorial improvements and corrections.
  - Updated reference [NIST-DH] to Revision 3 (2018).
- WG Version 01:
  - Add NIST P-256 as Cryptosuite 2.
  - Renumber message types.
  - Very minor editorial fixes.

Appendix I. Acknowledgments

Aleksi Peltonen modeled the protocol specification with the mCRL2 formal specification language. Shiva Prasad TP and Raghavendra MS implemented parts of the protocol with wpa_supplicant and hostapd. Their inputs helped us in improving the specification.

The authors would also like to thank Rhys Smith and Josh Howlett for providing valuable feedback as well as new use cases and requirements for the protocol. Thanks to Eric Rescorla, Darshak Thakore, Stefan Winter, Hannes Tschofenig, and Daniel Migault for interesting discussions in this problem space.

Authors' Addresses

Tuomas Aura
Aalto University
Aalto 00076
Finland

EMail: tuomas.aura@aalto.fi

Mohit Sethi
Ericsson
Jorvas 02420
Finland

EMail: mohit@piuha.net