

EMU
Internet-Draft
Intended status: Standards Track
Expires: May 20, 2021

M. Chen, Ed.
Li. Su
China Mobile
H. Wang
Huawei International Pte. Ltd.
November 16, 2020

Use Identity as Raw Public Key in EAP-TLS
draft-chen-emu-eap-tls-ibs-01

Abstract

This document specifies the use of identity as a raw public key in EAP-TLS both for TLS1.2 and TLS1.3, EAP-TLS for TLS1.2 is defined in RFC 5216 and EAP-TLS for TLS1.3 is defined in the draft draft-ietf-tls-dtls13. The protocol procedures of EAP-TLS-IBS will consistent with EAP-TLS's interactive process, Identity-based signature will be extended to support EAP-TLS's signature algorithms.

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 May 20, 2021.

Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must

include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1.	Introduction	2
2.	Terminology	4
3.	Structure of the Raw Public Key Extension	4
4.	EAP-TLS using raw public keys	6
4.1.	EAP TLS1.2 Client and Server Handshake Behavior	6
4.1.1.	raw public keys TLS exchange	6
4.1.2.	EAP-TLS handshake in TLS1.2	7
4.1.3.	raw public keys EAP-TLS exchange	8
4.1.4.	EAP-TLS1.2-IBS example	10
4.2.	EAP TLS1.3 Client and Server Handshake Behavior	12
4.2.1.	TLS1.3 handshake	12
4.2.2.	EAP-TLS1.3 handshake procedure	13
4.2.3.	raw public keys EAP-TLS1.3 exchange	14
4.2.4.	EAP-TLS1.3-IBS example	15
5.	Security Considerations	18
6.	IANA Considerations	18
7.	Acknowledgement	18
8.	References	18
8.1.	Normative References	18
8.2.	Informative references	19
	Authors' Addresses	19

1. Introduction

The Extensible Authentication Protocol (EAP) defined in RFC 3748 [RFC3748] can provide support for multiple authentication methods. Transport Layer Security (TLS) provides for mutual authentication, integrity-protected ciphersuite negotiation, and exchange between two endpoints. The EAP-TLS defined in RFC 5216 [RFC5216] which combines EAP and TLS that apply EAP method to load TLS procedures.

Traditionally, TLS client and server public keys are obtained in PKIX containers in-band as part of the TLS handshake procedure and are validated using trust anchors based on a PKIX certification authority (CA). But there is another method, Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) are defined in RFC 7250 [RFC7250], the document defines two TLS extensions `client_certificate_type` and `server_certificate_type`, which can be used as part of an extended TLS handshake when raw public keys are used. In the draft `draft-ietf-emu-eap-tls13` reads certificates can be of any type supported by TLS including raw public keys. In

RFC7250[RFC7250] it assuming that an out-of-band mechanism is used to bind the public key to the entity presenting the key.

Digital signatures provide the functions of Sender reliability and Message integrity. A chain of trust for such signatures is usually provided by certificates, but in low-bandwidth and resource-constrained environments, the use of certificates might be undesirable. In comparison with the original certificate, the raw public key is fairly small. This document describes a signature algorithm using identity as a raw public key in EAP-TLS, instead of transmitting a full certificate in the EAP-TLS message, only public keys are exchanged between client and server, also known as EAP-TLS-IBS.

With the existing raw public key scheme, a public key and identity mapping table is required at server. This table usually established with offline method and may require additional efforts for establishment and maintenance, especially when the number of devices are huge. On the other hand, with IBS signature algorithm, it not only can take the advantage of raw public key, but also eliminates the efforts for the mapping table establishment and maintenance at the server side. Instead, a small table for CRL is enough for exclude revoked identity from accessing the network. A number of IBE and IBS algorithms have been standardized, such as ECCSI defined in RFC 6507[RFC6507].

IBC was first proposed by Adi Shamir in 1984. For an IBC system, a Key Management System (KMS) is required to generate keys for devices. The KMS choose its KMS Secret Authentication Key(KSAK) as the root of trust. A public parameter, KMS Public Authentication Key (KPAK) is derived from this secrete key and is used by others in verifying the signature. The signatures are generated by an entity with private keys obtained from the KMS. KMS is a trusted third party, users or devices can obtain private key using their identities from KMS. In IBS the private key is also known as Secret Signing Key(SSK). A sender can sign a message using SSK. The receiver can verify the signature with sender's identity and the KPAK.

This document is organized as follows: the second section defines the terms used in the text; the third section gives a brief overview of the IBS algorithms; the fourth section presents the example message flow and message format for EAP-TLS-IBS and follows by security consideration and IANA cosideration etc.

2. Terminology

The readers should be familiar with the terms defined in.

In addition, this document makes use of the following terms:

IBC: Identity-Based Cryptograph, it is an asymmetric public key cryptosystem.

IBS: Identity-based Signature, such as ECCSI.

PKI: Public Key Infrastructure, an infrastructure built with a public-key mechanism.

authenticator: The entity initiating EAP authentication.

peer: The entity that responds to the authenticator.

backend authenticator server: A backend authentication server is an entity that provides an authentication service to an authenticator. When used, this server typically executes EAP methods for the authenticator.

EAP 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. Structure of the Raw Public Key Extension

To support the negotiation of using raw public between client and server, a new certificate structure is defined in RFC 7250[RFC7250]. It is used by the client and server in the hello messages to indicate the types of certificates supported by each side. When RawPublicKey type is selected for authentication, SubjectPublicKeyInfo which is a data structure is used to carry the raw public key and its cryptographic algorithm.

The SubjectPublicKeyInfo structure is defined in Section 4.1 of RFC 5280 [PKIX][RFC5280] and not only contains the raw keys, such as the public exponent and the modulus of an RSA public key, but also an algorithm identifier. The algorithm identifier can also include parameters. The structure of SubjectPublicKeyInfo is shown in Figure 1:

```
SubjectPublicKeyInfo ::= SEQUENCE {
    algorithm           AlgorithmIdentifier,
    subjectPublicKey    BIT STRING }

AlgorithmIdentifier ::= SEQUENCE {
    algorithm           OBJECT IDENTIFIER,
    parameters         ANY DEFINED BY algorithm OPTIONAL }
```

Figure 1: SubjectPublicKeyInfo ASN.1 Structure

The algorithms identifiers are Object Identifier(OIDs), AlgorithmIdentifier is also data structure with two fields, OID represent the cryptographic algorithm used with raw public key, such as ECCSI, parameters are the necessary parameters associated with the algorithm.

In the case of IBS algorithm, the User's identity is the raw public key which can be represented by "subjectPublicKey", when ECCSI is used as the Identity-based signature algorithm, then "algorithm" is for ECCSI, and "parameters" is the parameters needed in ECCSI.

So far, IBS has the following four algorithms, the following table is the corresponding table of Key type and OID.

Key Type	Document	OID
ISO/IEC 14888-3 IBS-1	ISO/IEC 14888-3: IBS-1 mechanism	1.0.14888.3.0.7
ISO/IEC 14888-3 IBS-2	ISO/IEC 14888-3: IBS-2 mechanism	1.0.14888.3.0.8
ISO/IEC 14888-3 ChineseIBS (SM9)	ISO/IEC 14888-3: ChineseIBS mechanism	1.2.156.10197.1.302.1
Elliptic Curve-Based Signatureless For Identity-based Encryption (ECCSI)	Section 5.2 in RFC 6507	1.3.6.1.5.5.7.6.29

Table 1: Algorithm Object Identifiers

In the draft draft-wang-tls-raw-public-key-with-ibc, there extend signature scheme with IBS algorithm which indicated in the client's "signature_algorithms" extension. The SignatureScheme data structure also keep pace with the section 4.

4. EAP-TLS using raw public keys

This section describes EAP-TLS-IBS both in the case of TLS1.2 and TLS1.3, each section contains EAP-TLS and EAP-TLS using raw public keys full message authentication, and finally give the example when using IBS.

4.1. EAP TLS1.2 Client and Server Handshake Behavior

4.1.1. raw public keys TLS exchange

As described in [RFC7250][RFC7250], the document intrudoces the use of raw public keys in TLS/DTLS, the basic raw public key TLS exchange will appear as follows, Figure 2 shows the client_certificate_type and server_certificate_type extensions added to the client and server hello messages. An extension type MUST NOT appear in the ServerHello unless the same extension type appeared in the corresponding ClientHello, defined in RFC5246[RFC5246].

The `server_certificate_type` extension in the client hello indicates the types of certificates the client is able to process when provided by the server in a subsequent certificate payload.

The `client_certificate_type` and `server_certificate_type` extensions sent in the client hello each carry a list of supported certificate types, sorted by client preference. When the client supports only one certificate type, it is a list containing a single element. Many types of certificates can be used, such as `RawPublicKey`, `X.509` and `OpenPGP`.

```

client_hello,
client_certificate_type,
server_certificate_type  ->

<- server_hello,
    client_certificate_type,
    server_certificate_type,
    certificate,
    server_key_exchange,
    certificate_request,
    server_hello_done

certificate,
client_key_exchange,
certificate_verify,
change_cipher_spec,
finished                ->

<- change_cipher_spec,
    finished

Application Data      <----->      Application Data

```

Figure 2: Basic Raw Public Key TLS Exchange

4.1.2. EAP-TLS handshake in TLS1.2

As described in [RFC3748] [RFC3748], the EAP-TLS conversation will typically begin with the authenticator and the peer negotiating EAP. The authenticator will then typically send an EAP-Request/Identity packet to the peer, and the peer will respond with an EAP-Response/Identity packet to the authenticator, containing the peer's user-Id. The authenticator MAY act as a pass-through device, with the EAP packets received from the peer being encapsulated for transmission to a backend authentication server.

In the case where the EAP-TLS mutual authentication is successful, defined in RFC5216 [RFC5216], the conversation will appear as follows:

Authenticating Peer	Authenticator
	<- EAP-Request/ Identity
EAP-Response/ Identity (MyID) ->	<- EAP-Request/ EAP-Type=EAP-TLS (TLS Start)
EAP-Response/ EAP-Type=EAP-TLS (TLS client_hello)->	<- EAP-Request/ EAP-Type=EAP-TLS (TLS server_hello, TLS certificate, TLS server_key_exchange, TLS certificate_request, TLS server_hello_done)
EAP-Response/ EAP-Type=EAP-TLS (TLS certificate, TLS client_key_exchange, TLS certificate_verify, TLS change_cipher_spec, TLS finished) ->	<- EAP-Request/ EAP-Type=EAP-TLS (TLS change_cipher_spec, TLS finished)
EAP-Response/ EAP-Type=EAP-TLS ->	<- EAP-Success

Figure 3: EAP-TLS authentication procedure with TLS1.2 handshake

4.1.3. raw public keys EAP-TLS exchange

This section describes EAP-TLS extend using raw public keys, the procedures is as follows, In the discussion, we will use the term "EAP server" to denote the ultimate endpoint conversing with the peer.

```

Authenticating Peer      EAP server
-----
EAP-Response/
Identity (MyID) ->
EAP-Response/
EAP-Type=EAP-TLS
(TLS client_hello
+signature_algorithm
server_certificate_type,
client_certificate_type)->
EAP-Response/
EAP-Type=EAP-TLS
(TLS certificate,
TLS client_key_exchange,
TLS certificate_verify,
TLS change_cipher_spec,
TLS finished) ->
EAP-Response/
EAP-Type=EAP-TLS ->

<- EAP-Request/
Identity
<- EAP-Request/
EAP-Type=EAP-TLS
(TLS Start)
<- EAP-Request/
EAP-Type=EAP-TLS
(TLS server_hello,
 {client_certificate_type}
 {server_certificate_type}
 {TLS certificate}
 {TLS server_key_exchange}
 {TLS certificate_request}
 {TLS server_hello_done}
 )
<- EAP-Request/
EAP-Type=EAP-TLS
(TLS change_cipher_spec,
 TLS finished)
<- EAP-Success

```

Figure 4: EAP-TLS extend raw public keys authentication procedure with TLS1.2 handshake

4.1.4. EAP-TLS1.2-IBS example

In this example, both the TLS client and server use ECCSI for authentication, and they are restricted in that they can only process ECCSI signature algorithm. As a result, the TLS client sets both the `server_certificate_type` and the `client_certificate_type` extensions to be raw public key; in addition, the client sets the signature algorithm in the client hello message to be `eccsi_sha256`.

```

Authenticating Peer                                EAP server
-----
EAP-Response/                                     <- EAP-Request/
Identity (MyID) ->                                Identity

EAP-Response/                                     <- EAP-Request/
EAP-Type=EAP-TLS                                  EAP-Type=EAP-TLS
(TLS client_hello                                 (TLS Start)
signature_algorithm = (eccsi_sha256)
server_certificate_type = (RawPublicKey,...)
client_certificate_type = (RawPublicKey,...))->

                                                    <- EAP-Request/
EAP-Type=EAP-TLS                                  EAP-Type=EAP-TLS
(TLS server_hello,                                (TLS server_hello,
{client_certificate_type = RawPublicKey}           {client_certificate_type = RawPublicKey}
{server_certificate_type = RawPublicKey}           {server_certificate_type = RawPublicKey}
{certificate = (1.3.6.1.5.5.7.6.29, hash           {certificate = (1.3.6.1.5.5.7.6.29, hash
value of ECCSIPublicParameters),                  value of ECCSIPublicParameters),
serverID})}                                         serverID})}
{certificate_request = (eccsi_sha256)}             {certificate_request = (eccsi_sha256)}
{server_hello_done}                                {server_hello_done}
                                                    )

EAP-Response/                                     <- EAP-Request/
EAP-Type=EAP-TLS                                  EAP-Type=EAP-TLS
({certificate = ((1.3.6.1.5.5.7.6.29,              (TLS finished)
hash value of ECCSIPublicParameters),
ClientID}),
{certificate_verify = (ECCSI-Sig-Value)},
{finished}) ->

EAP-Response/                                     <- EAP-Request/
EAP-Type=EAP-TLS ->                               EAP-Type=EAP-TLS
                                                    (TLS finished)

                                                    <- EAP-Success

```

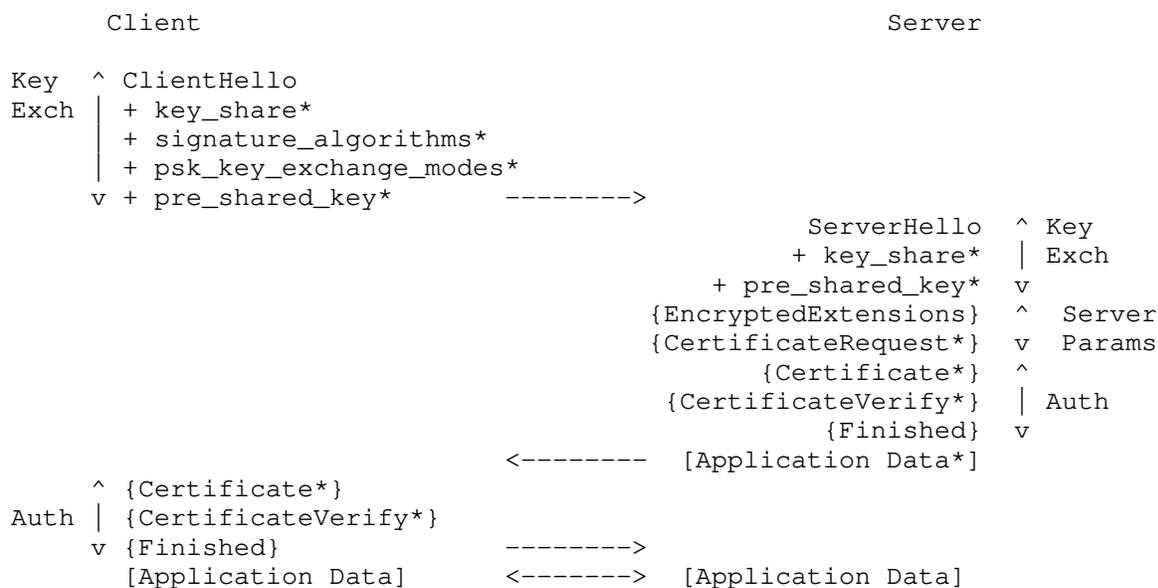
Figure 5: EAP-TLS1.2-IBS example

4.2. EAP TLS1.3 Client and Server Handshake Behavior

TLS1.3 defined in RFC8446, as TLS 1.3 is not directly compatible with previous versions, all versions of TLS incorporate a versioning mechanism which allows clients and servers to interoperably negotiate a common version if one is supported by both peers. when make the discussion on EAP-TLS using raw public keys we also make a different with TLS1.2, This section is for EAP-TLS1.3 handshake behavior using raw public keys and give example for EAP-TLS-IBS.

4.2.1. TLS1.3 handshake

TLS1.3 is more secure than TLS1.2 in preventing eavesdropping, tampering, and message forgery. The handshake can be thought of having three phases: Key Exchange, Server Parameters and Authentication. The message flow for full TLS handshake is as follows.



+ Indicates noteworthy extensions sent in the previously noted message.

* Indicates optional or situation-dependent messages/extensions that are not always sent.

{ } Indicates messages protected using keys derived from a [sender]_handshake_traffic_secret.

[] Indicates messages protected using keys derived from [sender]_application_traffic_secret_N.

Figure 6: Message Flow for Full TLS1.3 Handshake

4.2.2. EAP-TLS1.3 handshake procedure

EAP-TLS mutual authentication in the case of TLS1.3. defined in the draft-ietf-emu-eap-tls13. TLS 1.3 provides significantly improved security, privacy, and reduced latency when compared to earlier versions of TLS. EAP-TLS with TLS 1.3 further improves security and privacy by mandating use of privacy and revocation checking.

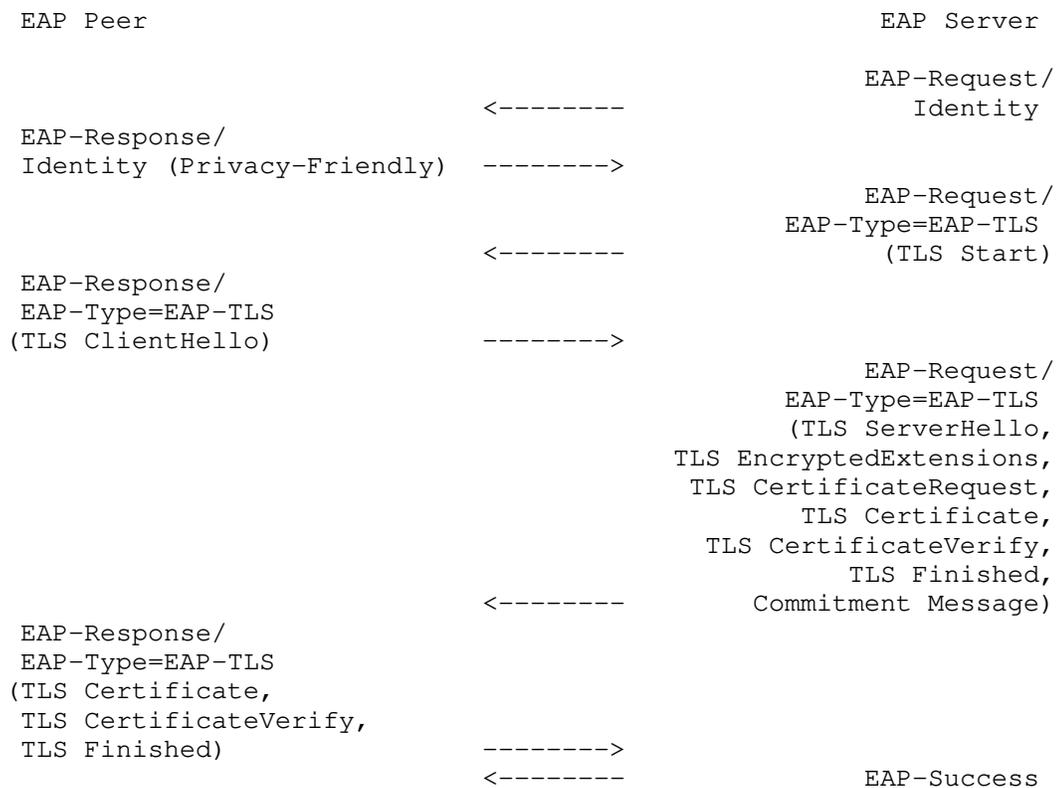


Figure 7: EAP-TLS mutual authentication with TLS1.3 handshake

4.2.3. raw public keys EAP-TLS1.3 exchange

This section describes EAP-TLS1.3 extend using raw public keys, the procedures is as follows, both client and server have the extension "key_share", the "key_share" extension contains the endpoint's cryptographic parameters. the "signature_algorithm" extension contains the signature algorithm and hash algorithms the client and server can support for the new signature algorithms specific to the IBS algorithms. When IBS is chosen as signature algorithm, the server need to indicated the required IBS signature algorithms int the signature_algorithm extension within the CertificateRequest.

```

Authenticating Peer      EAP server
-----
EAP-Response/
Identity (MyID) ->

EAP-Response/
EAP-Type=EAP-TLS
(TLS client_hello
+key_share
+signature_algorithm
server_certificate_type,
client_certificate_type)->

EAP-Response/
EAP-Type=EAP-TLS
({certificate}
{CertificateVerify}
{Finished}
[Application Data]) ->

<- EAP-Request/
Identity

<- EAP-Request/
EAP-Type=EAP-TLS
(TLS Start)

<- EAP-Request/
EAP-Type=EAP-TLS
(TLS server_hello,
+key_share
{EncryptedExtensions}
{client_certificate_type}
{server_certificate_type}
{certificate}
{CertificateVerify}
{certificateRequest}
{Finished}
[Application Data]
)

<- EAP-Success
[Application Data]

```

Figure 8: EAP-TLS1.3 authentication procedure with raw public keys

4.2.4. EAP-TLS1.3-IBS example

When the EAP server receives the client hello, it processes the message. Since it has an ECCSI raw public key from the KMS, it indicates that it agrees to use ECCSI and provides an ECCSI key by placing the SubjectPublicKeyInfo structure into the Certificate

payload back to the client, including the OID, the identity of server, ServerID, which is the public key of server also, and hash value of KMS public parameters. The `client_certificate_type` indicates that the TLS server accepts raw public key. The TLS server demands client authentication, and therefore includes a `certificate_request`, which requires the client to use `eccsi_sha256` for signature. A signature value based on the `eccsi_sha256` algorithm is carried in the `CertificateVerify`. The client, which has an ECCSI key, returns its ECCSI public key in the `Certificate` payload to the server, which includes an OID for the ECCSI signature. The example of EAP-TLS1.3-IBS is as follows:

```

Authenticating Peer                                EAP server
-----
EAP-Response/                                     <- EAP-Request/
Identity (MyID) ->                                Identity

EAP-Response/                                     <- EAP-Request/
EAP-Type=EAP-TLS                                  EAP-Type=EAP-TLS
(TLS client_hello                                 (TLS Start)
signature_algorithm = (eccsi_sha256)
server_certificate_type = (RawPublicKey)
client_certificate_type = (RawPublicKey))->

                                                    <- EAP-Request/
                                                    EAP-Type=EAP-TLS
                                                    (TLS server_hello,
                                                    +key_share
                                                    {client_certificate_type = RawPublicKey}
                                                    {server_certificate_type = RawPublicKey}
                                                    {certificate = (1.3.6.1.5.5.7.6.29, hash
                                                    value of ECCSIPublicParameters,
                                                    serverID)}
                                                    {certificate_request = (eccsi_sha256)}
                                                    {certificate_verify = {ECCSI-Sig-Value}}
                                                    {Finished}
                                                    [Application Data]
                                                    )

EAP-Response/                                     <- EAP-Success
EAP-Type=EAP-TLS                                  [Application Data]
({certificate = ((1.3.6.1.5.5.7.6.29,
hash value of ECCSIPublicParameters),
ClientID)},
{certificate_verify = (ECCSI-Sig-Value)},
{Finished})
[Application Data] ->

```

Figure 9: EAP-TLS1.3-IBS example

5. Security Considerations

TBD

6. IANA Considerations

This document registers the following item in the "Method Types" registry under the "extensible Authentication Protocol (EAP) Registry" heading.

Value	Description
TBD	EAP-TLS-IBS

7. Acknowledgement

TBD

8. References

8.1. Normative References

- [RFC3748] Aboba, B., Blunk, L., Vollbrecht, J., Carlson, J., and H. Levkowitz, Ed., "Extensible Authentication Protocol (EAP)", RFC 3748, DOI 10.17487/RFC3748, June 2004, <<https://www.rfc-editor.org/info/rfc3748>>.
- [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>>.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<https://www.rfc-editor.org/info/rfc5246>>.
- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <<https://www.rfc-editor.org/info/rfc5280>>.

[RFC7250] Wouters, P., Ed., Tschofenig, H., Ed., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7250, DOI 10.17487/RFC7250, June 2014, <<https://www.rfc-editor.org/info/rfc7250>>.

8.2. Informative references

[RFC6507] Groves, M., "Elliptic Curve-Based Certificateless Signatures for Identity-Based Encryption (ECCSI)", RFC 6507, DOI 10.17487/RFC6507, February 2012, <<https://www.rfc-editor.org/info/rfc6507>>.

Authors' Addresses

Meiling Chen (editor)
China Mobile
32, Xuanwumen West
BeiJing, BeiJing 100053
China

Email:
chenmeiling@chinamobile.com

Li Su
China Mobile

32, Xuanwumen West

BeiJing

100053

China

Email:
suli@chinamobile.com

Haiguang Wang
Huawei International Pte. Ltd.
11 North Buona Vista Dr, #17-08
Singapore 138589
SG

Phone: +65 6825 4200
Email: wang.haiguang1@huawei.com

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 14, 2021

O. Friel
Cisco
D. Harkins
Hewlett-Packard Enterprise
July 13, 2020

Bootstrapped TLS Authentication
draft-friel-tls-eap-dpp-01

Abstract

This document defines a TLS extension that enables a server to prove to a client that it has knowledge of the public key of a key pair where the client has knowledge of the private key of the key pair. Unlike standard TLS key exchanges, the public key is never exchanged in TLS protocol messages. Proof of knowledge of the public key is used by the client to bootstrap trust in the server. The use case outlined in this document is to establish trust in an EAP server.

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 January 14, 2021.

Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must

include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Bootstrap Key Pair	2
1.2. Alignment with Wi-Fi Alliance Device Provisioning Profile	3
2. Bootstrapping in TLS 1.3	4
2.1. Server Ephemeral Key Options	4
2.2. Bootstrap Key Extension	4
2.3. Changes to TLS 1.3 Handshake	5
2.4. Changes to TLS 1.3 Key Schedule	6
3. Using TLS Bootstrapping in EAP	7
4. Summary of Work	8
5. IANA Considerations	9
6. Security Considerations	9
7. Informative References	9
Authors' Addresses	10

1. Introduction

On-boarding of devices with no, or limited, user interface can be difficult. Typically, a credential is needed to access the network and network connectivity is needed to obtain a credential. This poses a catch-22.

If trust in the integrity of a device's public key can be obtained in an out-of-band fashion, a device can be authenticated and provisioned with a usable credential for network access. While this authentication can be strong, the device's authentication of the network is somewhat weaker. [Stajano]] presents a functional security model to address this asymmetry.

There are on-boarding protocols, such as [DPP], to address this use case but they have drawbacks. [DPP] for instance does not support wired network access. This document describes an on-boarding protocol, which we refer to as TLS Proof of Knowledge or TLS-POK.

1.1. Bootstrap Key Pair

The mechanism for on-boarding of devices defined in this document relies on bootstrap key pairs. A client device has an associated elliptic curve (EC) key pair. The key pair may be static and baked into device firmware at manufacturing time, or may be dynamic and generated at on-boarding time by the device. If this public key, specifically the ASN.1 SEQUENCE SubjectPublicKeyInfo from [RFC5280],

can be shared in a trustworthy manner with a TLS server, a form of "origin entity authentication" (the step from which all subsequent authentication proceeds) can be obtained.

The exact mechanism by which the server gains knowledge of the public key is out of scope of this specification, but possible mechanisms include scanning a QR code to obtain a base64 encoding of the ASN.1-formatted public key or upload of a Bill of Materials (BOM). If the QR code is physically attached to the client device, or the BOM is associated with the device, the assumption is that the public key obtained in this bootstrapping method belongs to the client. In this model, physical possession of the device implies legitimate ownership.

The server may have knowledge of multiple bootstrap public keys corresponding to multiple devices, and TLS extensions are defined in this document that enable the server to identify a specific bootstrap public key corresponding to a specific device.

Using the extensions defined herein, the client proves to the server that it has possession of the private analog to its public bootstrapping key. Provided that the mechanism in which the server obtained the bootstrapping key is trustworthy, a commensurate amount of authenticity of the resulting connection can be obtained. The server also proves that it knows the client's public key which, if the client does not gratuitously expose its public key, can be used to obtain a modicum of correctness, that the client is connecting to the correct network (see [Stajano]).

1.2. Alignment with Wi-Fi Alliance Device Provisioning Profile

The definition of the bootstrap public key aligns with that given in [DPP]. This, for example, enables the QR code format as defined in [DPP] to be reused for TLS-POK. Therefore, a device that supports both wired LAN and Wi-Fi LAN connections can have a single QR code printed on its label, and the bootstrap key can be used for DPP if the device bootstraps against a Wi-Fi network, or TLS-POK if the device bootstraps against a wired network. Similarly, a common bootstrap public key format could be imported in a BOM into a server that handles devices connecting over both wired and Wi-Fi networks.

Any bootstrapping method defined for, or used by, [DPP] is compatible with TLS-POK.

2. Bootstrapping in TLS 1.3

The bootstrapping modifications introduce an extension to identify a "bootstrapping" key into the TLS 1.3 handshake. This key MUST be from a cryptosystem suitable for doing (EC)DH. When using the bootstrapping extension, two (EC)DH operations are performed, a static-ephemeral (using the client's bootstrapping key and an ephemeral key generated by the server) and an ephemeral-ephemeral (using the client's ephemeral key and an ephemeral key generated by the server).

2.1. Server Ephemeral Key Options

[[TODO: Need to decide which mechanism to use here]]

There are two options for generation of the server's ephemeral key.

- o server generates one ephemeral key pair: the key pair is reused by the server for both static-ephemeral (bootstrap) and ephemeral-ephemeral (TLS key_share) ECDH exchanges. This means that the bootstrap and key_share keys MUST be on the same curve.
- o server generates two ephemeral key pairs: one that is used for the static-ephemeral (bootstrap) ECDH exchange, and one that is used for the ephemeral-ephemeral (TLS key_share) DH exchange. The bootstrap and key_share keys are independent. The key_share keys may be Finite Field DH keys.

This document assumes the latter and that the server generates unique ephemeral key pairs for bootstrap and key_share.

2.2. Bootstrap Key Extension

```
struct {
    select (Handshake.msg_type) {
        case client_hello:
            opaque bskey[32];

        case server_hello:
            opaque key_exchange<1..2^16-1>;
    };
} BootstrapKey;
```

The BootstrapKey extension is used by the client in its ClientHello message to specify its bootstrapping key identifier. The 'bskey' field of this extension SHALL consist of the base64 encoded SHA256 digest of the DER-encoded ASN.1 subjectPublicKeyInfo representation of the bootstrapping public key.

The BootstrapKey extension is used by the server in its ServerHello message to specify its ephemeral ECDH keying information. The 'key_exchange' field contains the key exchange information on the curve that the bootstrapping key is on.

2.3. Changes to TLS 1.3 Handshake

The client identifies the bootstrapping key in the ClientHello using the BootstrapKey extension 'bskey' field. The server looks up the client's bootstrapping key in its database by checking the SHA256 hash of each entry with the value received in the ClientHello. If no match is found, the server MAY proceed with the TLS handshake without using the bootstrapping key as input to the key schedule, or MAY terminate the TLS handshake with an alert.

[[TODO: should we define an explicit unknown_bsk_identity alert, similar to unknown_psk_identity]]

If the server found the matching bootstrap key, the server generates an ephemeral ECDH keypair on the curve indicated in the bootstrap public key information, and performs an ECDH operation using the client bootstrap key and the server's ephemeral keypair. The server includes a BootstrapKey extension in its ServerHello that includes its ephemeral ECDH public key in the 'key_exchange' field. This explicitly confirms to the client that the server has performed an ECDH operation using the bootstrap key, and has injected the output into the key schedule.

This is in addition to, and independent from, the (EC)DH that the server carries out when handling the key_share extension.

The handshake is shown in Figure 1.

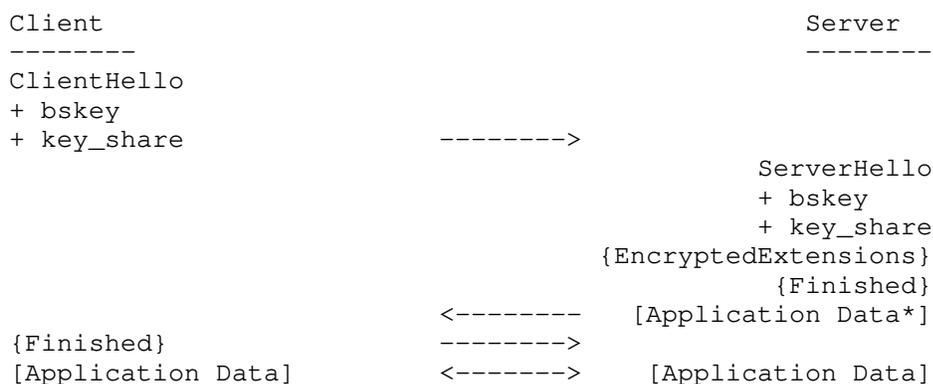


Figure 1: TLS 1.3 TLS-POK Handshake

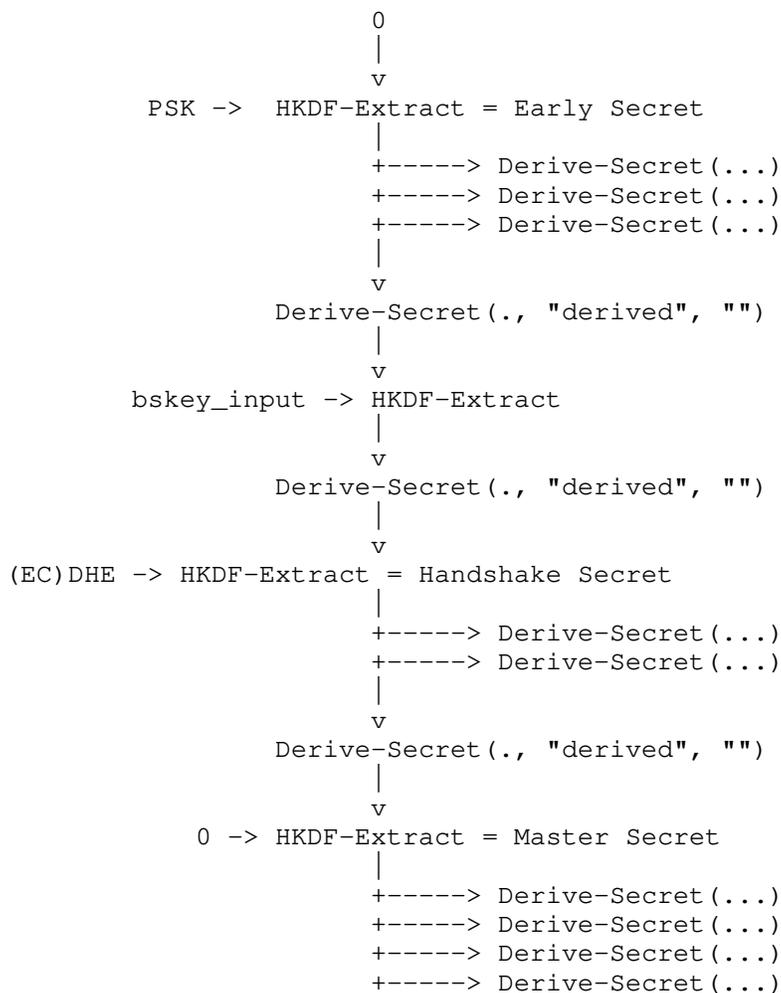
2.4. Changes to TLS 1.3 Key Schedule

[[TODO: The key schedule mechanism needs to closed.]]

Multiple options for modifying the TLS 1.3 key schedule have been proposed recently including [I-D.stebila-tls-hybrid-design] and [I-D.jhoyla-tls-extended-key-schedule]. The key schedule used for TLS-POK will align with the final direction chosen by the TLS WG.

This document proposes aligning with the model outlined in [I-D.jhoyla-tls-extended-key-schedule] where the shared secrets derived from the bskey and key_share key exchanges are injected in sequence into the key schedule.

The key schedule for TLS-POK is as follows:



3. Using TLS Bootstrapping in EAP

Enterprise deployments typically require an 802.1X/EAP-based authentication to obtain network access. Protocols like [RFC7030] can be used to enroll devices into a Certification Authority to allow them to authenticate using 802.1X/EAP. But this creates a Catch-22 where a certificate is needed for network access and network access is needed to obtain certificate.

Devices whose bootstrapping key can be obtained in an out-of-band fashion can perform an EAP-TLS-based exchange, for instance [RFC7170], and authenticate the TLS exchange using the bootstrapping extensions defined in Section 2. This network connectivity can then

be used to perform an enrollment protocol (such as provided by [RFC7170]) to obtain a credential for subsequent network connectivity and certificate lifecycle maintenance.

Upon "link up", an Authenticator on an 802.1X-protected port will issue an EAP Identify request to the newly connected peer. For unprovisioned devices that desire to take advantage of TLS-POK, there is no initial realm in which to construct an NAI (see [RFC4282]) so the initial EAP Identity response SHOULD contain simply the name "TLS-POK" in order to indicate to the Authenticator that an EAP method that supports TLS-POK SHOULD be started.

Authenticating Peer -----	Authenticator -----
	<- EAP-Request/ Identity
EAP-Response/ Identity (TLS-POK) ->	
	<- EAP-Request/ EAP-Type=TEAP (TLS Start)
	.
	.
	.

4. Summary of Work

[TODO: agree with WG chairs where this work lives and where it should be documented.]

The protocol outlined here can be broadly broken up into 3 distinct areas:

- o TLS extensions to transport the bootstrap public key identifier
- o TLS key schedule enhancements to inject bootstrap public key keying material
- o TEAP extensions to leverage the new TLS-POK handshake for trust establishment

This document captures all 3 areas, but it may be more appropriate to split the work into multiple documents e.g.:

- o piggy back on top of [I-D.jhojla-tls-extended-key-schedule] for TLS key schedule enhancements

- o include the TEAP extensions in [I-D.lear-eap-teap-brski] draft

5. IANA Considerations

IANA will allocated an ExtensionType for the bskey extension from the appropriate TLS 1.3 repository and replace TBD in this document with that number.

6. Security Considerations

Bootstrap and trust establishment by the TLS server is based on proof of knowledge of the client's bootstrap public key. An attack on the bootstrapping method which substitutes the public key of a corrupted device for the public key of an honest device can result in the TLS sever on-boarding and trusting the corrupted device.

Trust on the part of the client is not strong and is based on an assumption that the public bootstrapping key is not widely disseminated. If an adversary has knowledge of the bootstrap public key, the adversary may be able to make the client bootstrap against the adversary's network. For example, if an adversary intercepts and scans QR labels on clients, and the adversary can force the client to connect to its server, then the adversary can complete the TLS-POK handshake with the client and the client will connect to the adversary's server. Since physical possession implies ownership, there is nothing to prevent a stolen device from being on-boarded.

7. Informative References

[DPP] Wi-Fi Alliance, "Device Provisioning Profile", 2020.

[I-D.jhoyle-tls-extended-key-schedule]
Hoyland, J. and C. Wood, "TLS 1.3 Extended Key Schedule", draft-jhoyle-tls-extended-key-schedule-01 (work in progress), March 2020.

[I-D.lear-eap-teap-brski]
Lear, E., Friel, O., Cam-Winget, N., and D. Harkins, "TEAP Update and Extensions for Bootstrapping", draft-lear-eap-teap-brski-05 (work in progress), November 2019.

[I-D.stebila-tls-hybrid-design]
Stebila, D., Fluhrer, S., and S. Gueron, "Hybrid key exchange in TLS 1.3", draft-stebila-tls-hybrid-design-03 (work in progress), February 2020.

- [RFC4282] Aboba, B., Beadles, M., Arkko, J., and P. Eronen, "The Network Access Identifier", RFC 4282, DOI 10.17487/RFC4282, December 2005, <<https://www.rfc-editor.org/info/rfc4282>>.
- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <<https://www.rfc-editor.org/info/rfc5280>>.
- [RFC7030] Pritikin, M., Ed., Yee, P., Ed., and D. Harkins, Ed., "Enrollment over Secure Transport", RFC 7030, DOI 10.17487/RFC7030, October 2013, <<https://www.rfc-editor.org/info/rfc7030>>.
- [RFC7170] Zhou, H., Cam-Winget, N., Salowey, J., and S. Hanna, "Tunnel Extensible Authentication Protocol (TEAP) Version 1", RFC 7170, DOI 10.17487/RFC7170, May 2014, <<https://www.rfc-editor.org/info/rfc7170>>.
- [Stajano] Anderson, S., "The Ressurrecting Ducking: Security Issues for Ad-Hoc Wireless Networks", 1999.

Authors' Addresses

Owen Friel
Cisco

Email: ofriel@cisco.com

Dan Harkins
Hewlett-Packard Enterprise

Email: daniel.harkins@hpe.com

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 13, 2021

T. Aura
Aalto University
M. Sethi
Ericsson
July 12, 2020

Nimble out-of-band authentication for EAP (EAP-NOOB)
draft-ietf-emu-eap-noob-02

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 January 13, 2021.

Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents

(<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

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.	Cloning of Devices	41
6.3.	Identifying correct endpoints	41

6.4.	Trusted path issues and misbinding attacks	42
6.5.	Peer identifiers and attributes	43
6.6.	Identity protection	44
6.7.	Downgrading threats	44
6.8.	Recovery from loss of last message	45
6.9.	EAP security claims	46
7.	References	48
7.1.	Normative references	48
7.2.	Informative references	49
Appendix A.	Exchanges and events per state	52
Appendix B.	Application-specific parameters	53
Appendix C.	ServerInfo and PeerInfo contents	54
Appendix D.	EAP-NOOB roaming	56
Appendix E.	OOB message as URL	57
Appendix F.	Example messages	58
Appendix G.	TODO list	62
Appendix H.	Version history	62
Appendix I.	Acknowledgments	65
Authors' Addresses	66

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.

To summarize, devices may have the following characteristics:

- o no pre-established relation with the intended server or user,
- o no pre-provisioned device identifier or authentication credentials,
- o 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. The OOB messages in this specification are dynamically generated. We thus do not support static printed registration codes. One reason for requiring dynamic OOB messages is that the receipt of the OOB message authorizes the server to take ownership of the device. This is more secure than a static printed code, which could be leaked and later misused.

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.)

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.

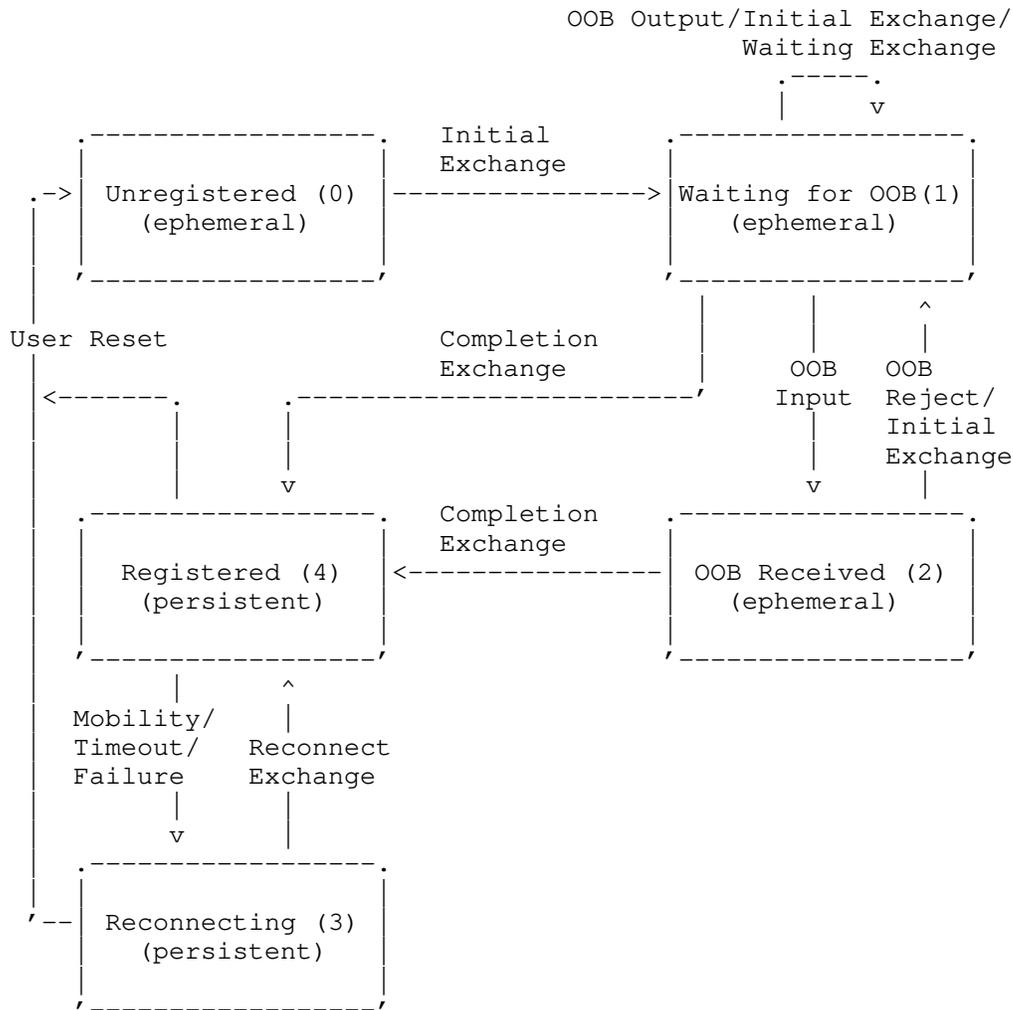


Figure 1: EAP-NOOB server-peer association state machine

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 a timeout or error may cause one or both endpoints to go back to the Unregistered state so that the Initial Exchange is repeated. After the Completion Exchange has resulted in EAP-Success, the association 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 the ephemeral 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 MUST 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.arpa" 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 an Authentication, Authorization, Accounting (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. Moreover, EAP-NOOB supports peers with no input capability in the user interface (e.g., LED light bulbs). 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. 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.

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 fields 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.

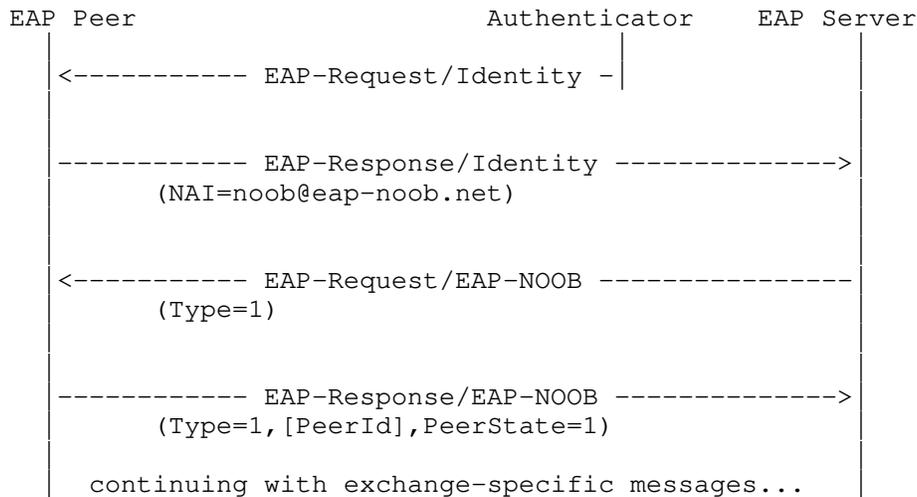


Figure 2: Common handshake in all EAP-NOOB exchanges

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.

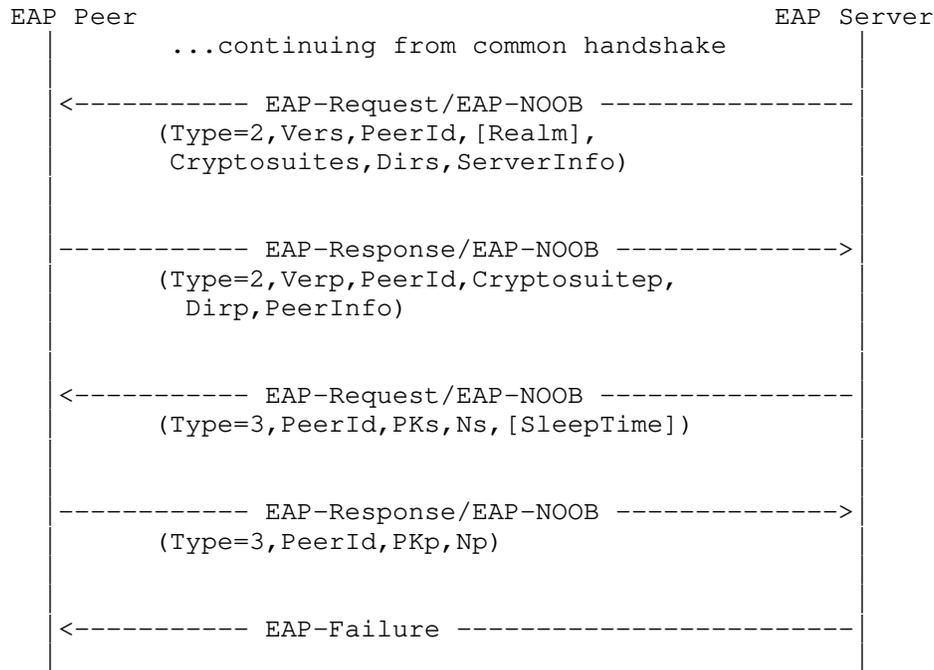


Figure 3: Initial Exchange

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 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.

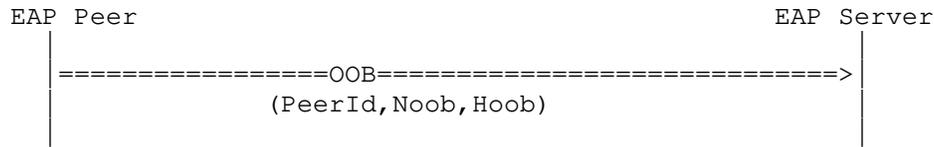


Figure 4: OOB Step, from peer to EAP server

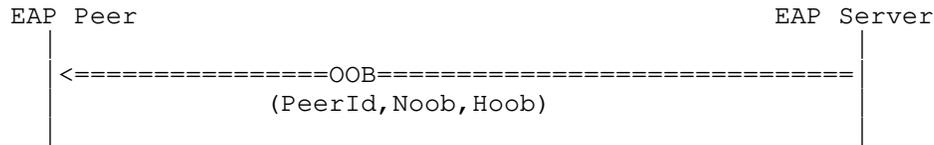


Figure 5: OOB Step, from EAP server to peer

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

the OOB receiver notifies the user about redundant OOB messages, but it MAY 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 $\text{NoobInterval} = \text{NoobTimeout} / 2$, 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 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 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.

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 K_{ms} and K_{mp} as defined in Section 3.5 and the message authentication codes MAC_s and MAC_p 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 MAC_s and the server finds that it has received the correct value of MAC_p , 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.

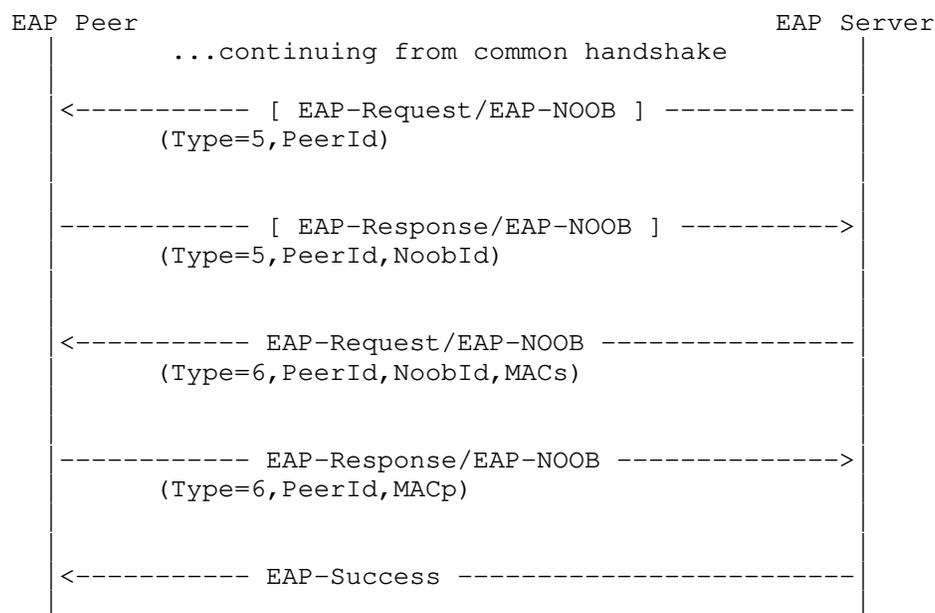


Figure 6: Completion Exchange

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 SHOULD 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 MUST 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

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 message authentication code. 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`.

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.

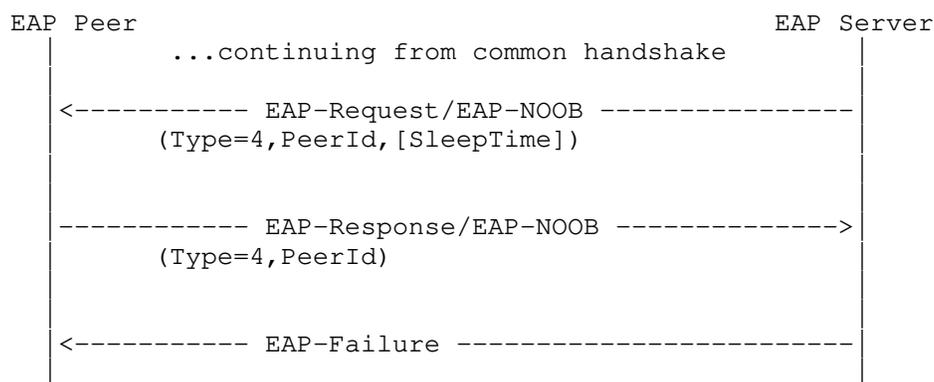


Figure 7: Waiting Exchange

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 ASCII string.

Another way to generate the identifiers is to choose a random 22-character alphanumeric ASCII 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. The peer MUST copy the PeerId byte-by-byte from the message where it was allocated, and the server MUST perform a byte-by-byte comparison between the received and the previously allocated PeerID. The peer 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.arpa". However, the peer MAY allow the user or application to provide a different default realm. 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.)

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.

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 detailed 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 (Dir=3), unless the type of the OOB channel limits them to one direction (Dir=1 or Dir=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, Cryptosuitep, Dirp, [Realm], PeerInfo, 0, PKs, Ns, PKp, Np, Noob) .
```

```
NoobId = H("NoobId", Noob) .
```

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

```
MACp = HMAC(Kmp; 1, Vers, Verp, PeerId, Cryptosuites, Dirs, ServerInfo, Cryptosuitep, 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], Np2, "")
```

```
MACp2 = HMAC(Kmp2; 1, Vers, Verp, PeerId, Cryptosuites, "", [ServerInfo], Cryptosuitep, "", [Realm], [PeerInfo], KeyingMode, [PKs2], Ns2, [PKp2], Np2, "")
```

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 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 ASCII characters)
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.arpa")	UTF-8 string
Kz	Persistent key material	32 bytes
KzPrev (at peer only)	Previous Kz value	32 bytes

Table 2: Persistent EAP-NOOB association

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 CryptosuitepPrev 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 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.

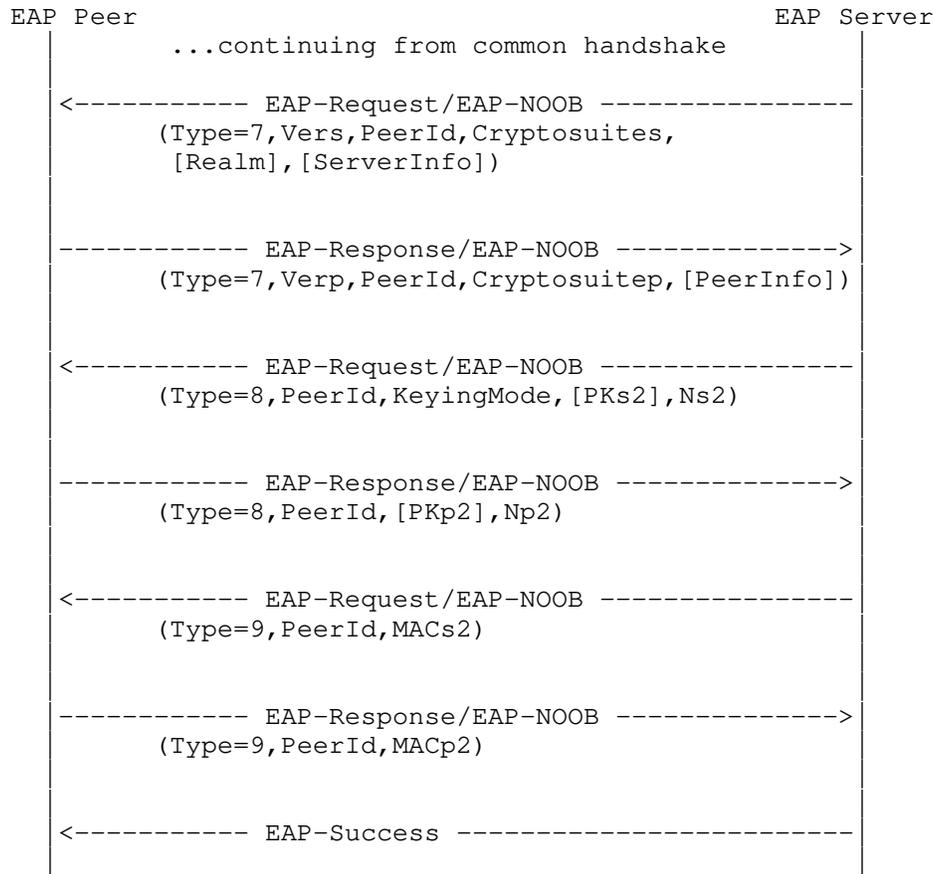


Figure 8: Reconnect Exchange

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

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 Completion	0..63	MSK	64
	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
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.

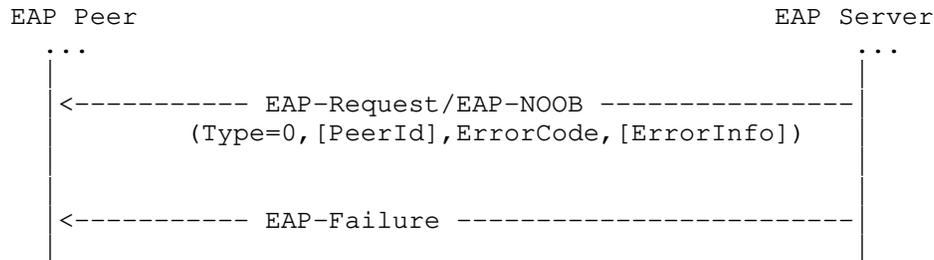


Figure 9: Error notification from server to peer

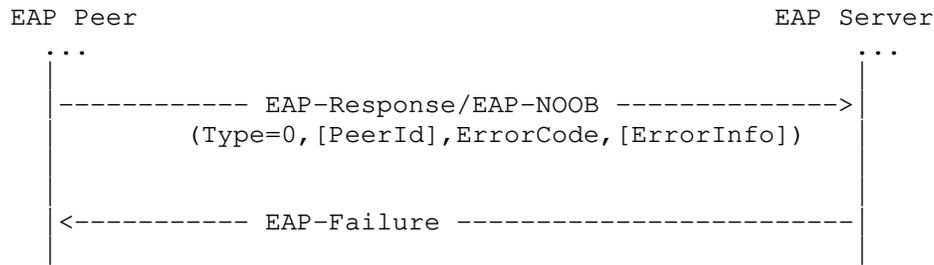


Figure 10: Error notification from peer to server

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. This range is reserved for "Private Use" and "Experimental Use", both locally and on the open Internet.

4.4. Domain name reservation considerations

The special-use domain "eap-noob.arpa" should be registered in the .arpa registry (<<https://www.iana.org/domains/arpa>>). The considerations required by [RFC6761] for registering this special-use domain name are the following:

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

- o Caching DNS Servers: Caching servers are not expected to recognize this domain name as special.
- o Authoritative DNS Servers: Authoritative DNS servers MUST respond to queries for "eap-noob.arpa" with NXDOMAIN.
- o DNS Server Operators: Except for the authoritative DNS server, there are no special requirements for the operators.
- o 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

- o Responsible Organization: Aalto University
- o Location: <<https://github.com/tuomaura/eap-noob>>
- o 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.
- o Level of Maturity: Alpha
- o Version compatibility: Version 06 of the draft implemented
- o Licensing: BSD
- o Contact Information: Tuomas Aura, tuomas.aura@aalto.fi

5.2. Implementation on Contiki

- o Responsible Organization: University of Murcia and Aalto University
- o Location: <<https://github.com/eduingles/coap-eap-noob>>
- o 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.
- o Level of Maturity: Alpha
- o Version compatibility: Version 05 of the draft implemented
- o Licensing: BSD
- o 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 [mcr12]. The model <<https://github.com/tuomaura/eap-noob/tree/master/protocolmodel/mcr12>> 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 disclosure of the OOB message, and such applications benefit from the second authentication feature.

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. Cloning of Devices

6.3. 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. A certificate or an attestation token (e.g., [I-D.tschofenig-tls-cwt] and [I-D.ietf-rats-eat]) 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.

After the device registration, an attacker could clone the device identity by copying the keys from the persistent EAP-NOOB association into another device. The attacker can be an outsider who gains access to the keys or the device owner who wants to have two devices matching the same registration. The cloning threats can be mitigated by creating the cryptographic keys and storing the persistent EAP-NOOB association on the peer device in a secure hardware component such as a trusted execution environment (TEE). Furthermore, remote attestation on the application level could provide assurance to the server that the device has not been cloned.

6.4. 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 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.

6.5. 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 on the application level, the PeerInfo is not authenticated information and should not be relied on.

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 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).

6.6. 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.

6.7. 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 K_z 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 K_z . EAP-NOOB protects against such attacks by allowing cryptosuite upgrades in the Reconnect Exchange and by updating the shared key material K_z 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.

6.8. 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 K_z 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.

6.9. 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

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. Levkowitz, 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.
- [I-D.ietf-rats-eat] Mandyam, G., Lundblade, L., Ballesteros, M., and J. O'Donoghue, "The Entity Attestation Token (EAT)", draft-ietf-rats-eat-03 (work in progress), February 2020.
- [I-D.tschofenig-tls-cwt] Tschofenig, H. and M. Brossard, "Using CBOR Web Tokens (CWTs) in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", draft-tschofenig-tls-cwt-01 (work in progress), November 2019.
- [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>>.
- [RFC5903] Fu, D. and J. Solinas, "Elliptic Curve Groups modulo a Prime (ECP Groups) for IKE and IKEv2", RFC 5903, DOI 10.17487/RFC5903, June 2010, <<https://www.rfc-editor.org/info/rfc5903>>.
- [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 3	Initial Exchange -	both 1 (0 on error) no change, notify user
server state: Waiting for OOB (1)		
0 1 2 3	Initial Exchange Waiting Exchange Completion Exchange -	both 1 (0 on error) both 1 (no change on error) both 4 (A) no change, notify user
server state: OOB Received (2)		
0 1 2 3	Initial Exchange Completion Exchange Completion Exchange -	both 1 (0 on error) both 4 (B) both 4 (A) no change, notify user
server state: Reconnecting (3) or Registered (4)		
0..2 3	- Reconnect Exchange	no change, notify user 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.
ServerInfoType	The value of the Type field and the other required fields in ServerInfo
PeerInfoType	The value of the Type field and the other required fields 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. Further specifications may specify domain-specific ServerInfo and PeerInfo contents.

Data field	Description
Type	Type-tag string that can be used by the peer as a hint for how to interpret the ServerInfo contents.
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 IEEE 802.11 wireless network identifier (SSID) strings used for roaming support, as suggested in Appendix D. JSON array of ASCII encoded SSID strings.
Base64SSIDList	List of IEEE 802.11 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.

Data field	Description
Type	Type-tag string that can be used by the server as a hint for how to interpret the PeerInfo contents.
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 be in upper or lower case and MAY include additional colon ':' or dash '-' characters that MUST be ignored by the server.
SSID	IEEE 802.11 network SSID for channel binding. The SSID is a ASCII string.
Base64SSID	IEEE 802.11 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 be in upper or lower case and MAY include additional colon ':' or dash '-' characters that MUST be ignored by the server.

Table 12: Suggested PeerInfo data fields

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 field 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.

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, so that there is a secure connection to the server and the man-in-the-middle attacker cannot read or modify the OOB message.

The ServerInfo in this case includes a field called ServerUrl of the following format with RECOMMENDED length of 60 characters or less:

```
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 ASCII string. The peer base64url encodes, without padding, the 16-byte values Noob and Hoob into 22-character ASCII 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-F4EfcU8D9ljxX_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.arpa
```

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":"noob.example.com","Cryptosuites":[1,2],"Dirs":3,"ServerInfo":{"Type":"url_wifi","Name":"Example","Url":"https://noob.example.com/sendOOB"}}
```

EAP response (type 2):

```
{"Type":2,"Verp":1,"PeerId":"07KRU6OgqX0HIeRFldnbSW","Cryptosuitep":1,"Dirp":2,"PeerInfo":{"Type":"wifi","Make":"Acme","Serial":"DU-9999","SSID":"Noob1","BSSID":"6c:19:8f:83:c2:80"}}
```

EAP request (type 3):

```
{"Type":3,"PeerId":"07KRU6OgqX0HIeRFldnbSW","PKs":{"kty":"EC","crv":"X25519","x":"hSDwCYkwp1R0i33ctD73Wg2_Og0mOBr066SpjqbTmo"},"Ns":"PYO7NVd9Af3BxEri1MI6hL8Ck49YxwCjSRPqlC1SPbw","SleepTime":60}
```

EAP response (type 3):

```
{"Type":3,"PeerId":"07KRU6OgqX0HIeRFldnbSW","PKp":{"kty":"EC","crv":"X25519","x":"3p7bfXt9wbTTW2HC7OQ1Nz-DQ8hbeGdNrfx-FG-IK08"},"Np":"HIvB6g0n2btpxEcuY7YXnWB-451ED6L6veQQd6ugiPFU"}
```

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

Identity response:

noob@noob.example.com

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=W3-icQoP19bJzJ
zI3kvpTA

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

Identity response:

noob@noob.example.com

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":"U00HwYGCS4nE
kzk2TPIE6g"}

EAP request (type 6):

{"Type":6,"PeerId":"07KRU6OgqX0HIeRFldnbSW","NoobId":"U00HwYGCS4nE
kzk2TPIE6g","MACs":"yQTuMzkEBb1sJCTUCTqHTbOR5Z2fs7s1tqMeCFqQT0k"}

EAP response (type 6):

{"Type":6,"PeerId":"07KRU6OgqX0HIeRFldnbSW","MACp":"MZOEImalmHZUL2
_kJX8KMHGDBNkvwqamiv8cCjxIZgM"}

```
===== Reconnect Exchange (KeyingMode = 1) =====

Identity response:
  noob@noob.example.com

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","Cryptosuites":
  [1,2],"Realm":"noob.example.com","ServerInfo":{"Type":"url_wifi","Name":
  "Example","Url":"https://noob.example.com/sendOOB"}}

EAP response (type 7):
  {"Type":7,"Verp":1,"PeerId":"07KRU6OgqX0HIeRFldnbSW","Cryptosuitep
  ":1,"PeerInfo":{"Type":"wifi","Make":"Acme","Serial":"DU-9999","SSID":
  "Noob1","BSSID":"6c:19:8f:83:c2:80"}}

EAP request (type 8):
  {"Type":8,"PeerId":"07KRU6OgqX0HIeRFldnbSW","KeyingMode":1,"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":"WsZU_JZNhE1ju
  dcVeEbIhiDEo1OXXOAxqxxgEP2FckI"}

EAP response (type 9):
  {"Type":9,"PeerId":"07KRU6OgqX0HIeRFldnbSW","MACp2":"sopB5xkLBjZBC
  81GE3YXdBDDadqRFh1vQTGIBXnr_Dk"}

===== Reconnect Exchange (KeyingMode = 2) =====

Identity response:
  noob@noob.example.com

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","Cryptosuites":[1,2],"Realm":"noob.example.com","ServerInfo":{"Type":"url_wifi","Name":"Example","Url":"https://noob.example.com/sendOOB"}}
```

```
EAP response (type 7):
  {"Type":7,"Verp":1,"PeerId":"07KRU6OgqX0HIeRFldnbSW","Cryptosuitep":1,"PeerInfo":{"Type":"wifi","Make":"Acme","Serial":"DU-9999","SSID":"Noob1","BSSID":"6c:19:8f:83:c2:80"}}
```

```
EAP request (type 8):
  {"Type":8,"PeerId":"07KRU6OgqX0HIeRFldnbSW","KeyingMode":2,"PKs2":{"kty":"EC","crv":"X25519","x":"hSDwCYkwp1R0i33ctD73Wg2_Og0mOBr066SpjqqbTmo"},"Ns2":"RDLahHBLIgmL_F_xcynrHurLPkCsrp3G3B_S82WUF4"}
```

```
EAP response (type 8):
  {"Type":8,"PeerId":"07KRU6OgqX0HIeRFldnbSW","PKp2":{"kty":"EC","crv":"X25519","x":"3p7bfXt9wbTTW2HC7OQ1Nz-DQ8hbeGdNrfx-FG-IK08"},"Np2":"jN0_V4P0JoTqwI9VHHQKd9ozUh7tQdc9ABd-j6oTy_4"}
```

```
EAP request (type 9):
  {"Type":9,"PeerId":"07KRU6OgqX0HIeRFldnbSW","MACs2":"CJ6bIYcSMvWD6NIqlzikdnmwVlvOJBJ3TODkg7QC140"}
```

```
EAP response (type 9):
  {"Type":9,"PeerId":"07KRU6OgqX0HIeRFldnbSW","MACp2":"TUP9IqEK78y4QfmsjDIijlLHHq3EgcQ_mYhXT0LSpmk"}
```

===== Reconnect Exchange (KeyingMode = 3) =====

These message examples show the case where the peer upgrades to Cryptosuite 2 during the Reconnect Exchange. The messages are generated with NIST P-256 ECDHE test vectors specified in section 8.1 of [RFC5903] (server=responder, peer=initiator).

```
Identity response:
  noob@noob.example.com
```

```
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","Cryptosuites":[1,2],"Realm":"noob.example.com","ServerInfo":{"Type":"url_wifi","Name":"Example","Url":"https://noob.example.com/sendOOB"}}
```

```
EAP response (type 7):
  {"Type":7,"Verp":1,"PeerId":"07KRU6OgqX0HIeRFldnbSW","Cryptosuitep":2,"PeerInfo":{"Type":"wifi","Make":"Acme","Serial":"DU-9999","SSID":"Noob1","BSSID":"6c:19:8f:83:c2:80"}}
```

```
EAP request (type 8):
  {"Type":8,"PeerId":"07KRU6OgqX0HIeRFldnbSW","KeyingMode":3,"PKs2":{"kty":"EC","crv":"P-256","x":"2tC2U5QiHPmwUeH-yleH0Jjf5j8kLnvlF0MN3JYEYA","y":"UnGgRhzbglLWHxxFb6PlmrH0WzOsz19Y0J4Fd7iZC7M"},"Ns2":"RDLahHBlIgnmL_F_xcynrHurLPkCsrp3G3B_S82WUF4"}
```

```
EAP response (type 8):
  {"Type":8,"PeerId":"07KRU6OgqX0HIeRFldnbSW","PKp2":{"kty":"EC","crv":"P-256","x":"0S37UonI1PgSCLcCcDmMNCKWlwoLzLdMc2_HVUSUv2M","y":"VvvzyjZswj6BV4VME8WNaqwj8Eatow-DU-dPMwOYcqs"},"Np2":"jn0_V4P0JoTqwI9VHHQKd9ozUh7tQdc9ABd-j6oTy_4"}
```

```
EAP request (type 9):
  {"Type":9,"PeerId":"07KRU6OgqX0HIeRFldnbSW","MACs2":"Baj0VovNadExDWhuY-ehTy92AgV236Xilx-hcjuePOs"}
```

```
EAP response (type 9):
  {"Type":9,"PeerId":"07KRU6OgqX0HIeRFldnbSW","MACp2":"8NZNZJqaptSsBA8CdCrFTfx1gBgdsdqW28oAAG27LpM"}
```

Appendix G. TODO list

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

Appendix H. Version history

- o Version 01:
 - * Fixed Reconnection Exchange.
 - * URL examples.
 - * Message examples.
 - * Improved state transition (event) tables.
- o 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.
- o 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.
- o Version 07:
 - * Updated example messages.

- * Added pointers to new implementation in Contiki.
- o Version 08:
 - * Editorial improvements and corrections.
- o WG Version 00:
 - * Editorial improvements and corrections.
 - * Updated reference [NIST-DH] to Revision 3 (2018).
- o WG Version 01:
 - * Add NIST P-256 as Cryptosuite 2.
 - * Renumber message types.
 - * Very minor editorial fixes.
- o WG Version 02:
 - * Updated message examples with all KeyingModes.
 - * Many editorial fixes and other updates based on the IoT directorate review of Dave Thaler.
 - * Text on cloning attacks based on review by Hannes Tschofenig.

Appendix I. Acknowledgments

Aleksi Peltonen modeled the protocol specification with the mCRL2 formal specification language. Max Crone, 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 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.

We would also like to express our sincere gratitude to Dave Thaler for his thorough review of the document.

Authors' Addresses

Tuomas Aura
Aalto University
Aalto 00076
Finland

EMail: tuomas.aura@aalto.fi

Mohit Sethi
Ericsson
Jorvas 02420
Finland

EMail: mohit@piuha.net

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: June 16, 2021

T. Aura
Aalto University
M. Sethi
Ericsson
A. Peltonen
Aalto University
December 13, 2020

Nimble out-of-band authentication for EAP (EAP-NOOB)
draft-ietf-emu-eap-noob-03

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, speaker 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 June 16, 2021.

Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

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 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.	Implementation with wpa_supplicant and hostapd	39
5.4.	Protocol modeling	39
6.	Security considerations	40

6.1.	Authentication principle	40
6.2.	Identifying correct endpoints	41
6.3.	Trusted path issues and misbinding attacks	43
6.4.	Peer identifiers and attributes	43
6.5.	Identity protection	44
6.6.	Downgrading threats	45
6.7.	Recovery from loss of last message	46
6.8.	EAP security claims	47
7.	References	49
7.1.	Normative references	49
7.2.	Informative references	50
Appendix A.	Exchanges and events per state	53
Appendix B.	Application-specific parameters	54
Appendix C.	ServerInfo and PeerInfo contents	55
Appendix D.	EAP-NOOB roaming	57
Appendix E.	OOB message as URL	58
Appendix F.	Example messages	59
Appendix G.	Version history	63
Appendix H.	Acknowledgments	66
Authors' Addresses	67

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.

To summarize, devices may have the following characteristics:

- o no pre-established relation with the intended server or user,
- o no pre-provisioned device identifier or authentication credentials,

- o 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 essential to minimize the length of the manually transferred or compared codes. The OOB messages in this specification are dynamically generated. We thus do not support static printed registration codes. One reason for requiring dynamic OOB messages is that the receipt of the OOB message authorizes the server to take ownership of the device. This is more secure than a static printed code, which could be leaked and later misused.

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.)

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.

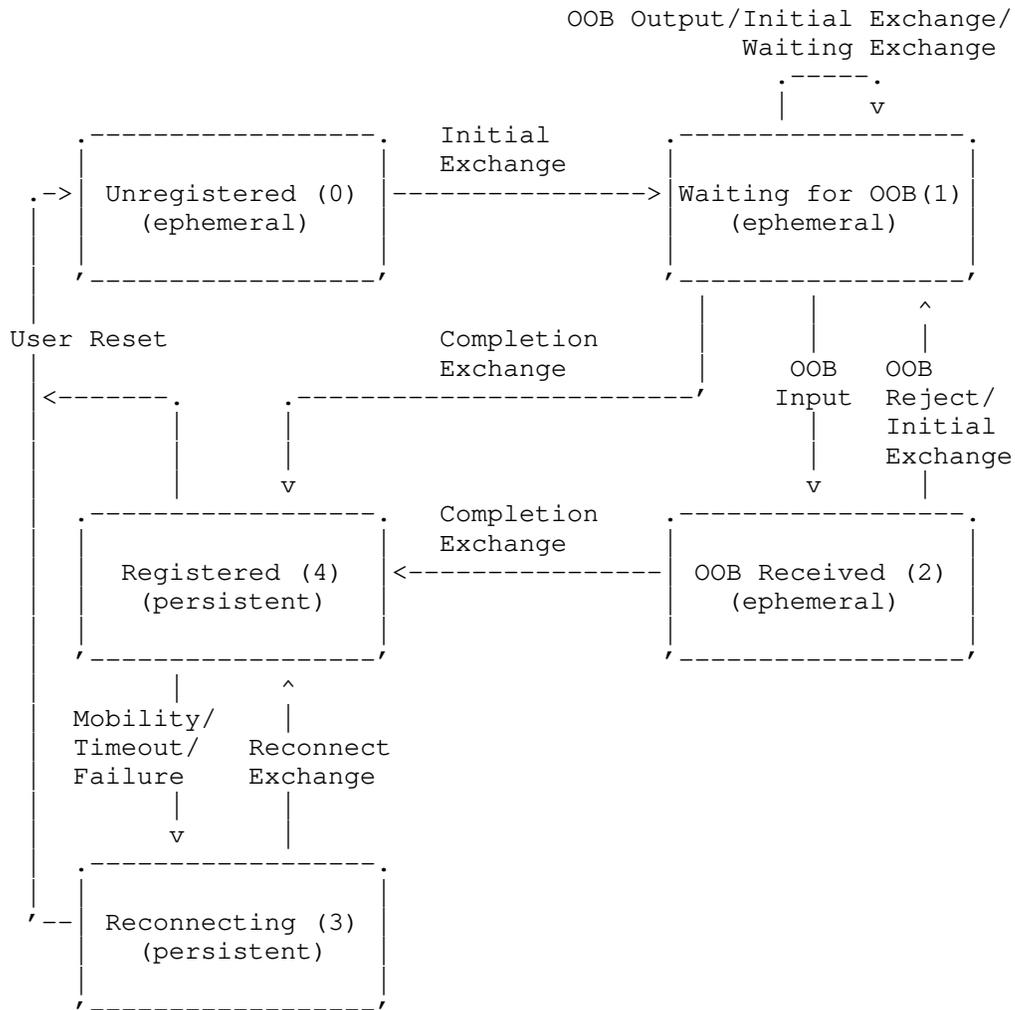


Figure 1: EAP-NOOB server-peer association state machine

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 a timeout or error may cause one or both endpoints to go back to the Unregistered state so that the Initial Exchange is repeated. After the Completion Exchange has resulted in EAP-Success, the association 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 the ephemeral 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 MUST 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.arpa" 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 an Authentication, Authorization and Accounting (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. Moreover, EAP-NOOB supports peers with no input capability in the user interface (e.g., LED light bulbs). Since users cannot initiate the protocol in these devices, the devices 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. 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.

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 fields 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 begins 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.

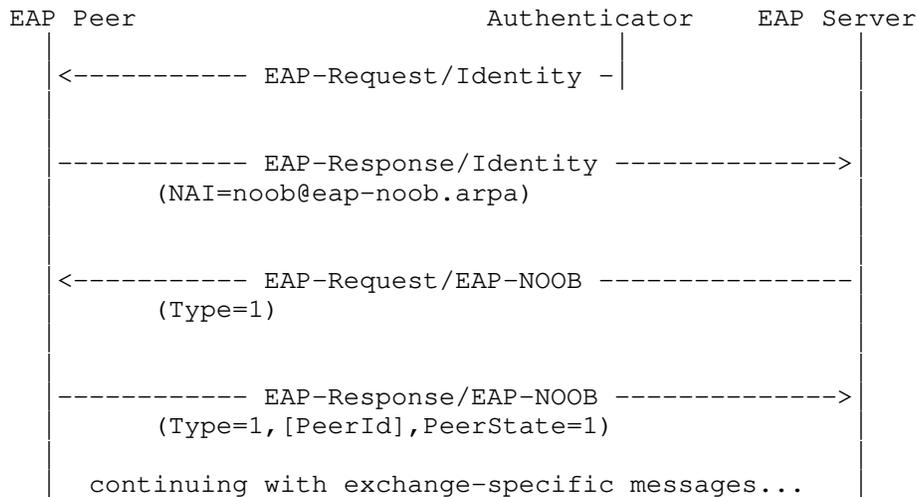


Figure 2: Common handshake in all EAP-NOOB exchanges

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 NewNAI for assigning a new NAI to the peer. The server allocates a new PeerId in the Initial Exchange regardless of any old PeerId received in the previous response (Type=1). 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.

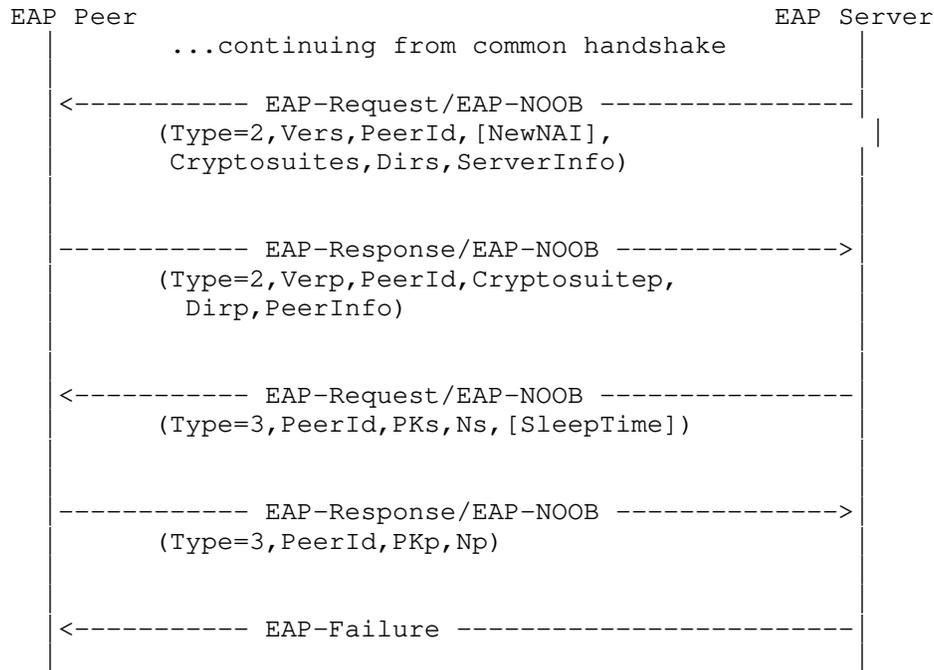


Figure 3: Initial Exchange

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 as 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 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.

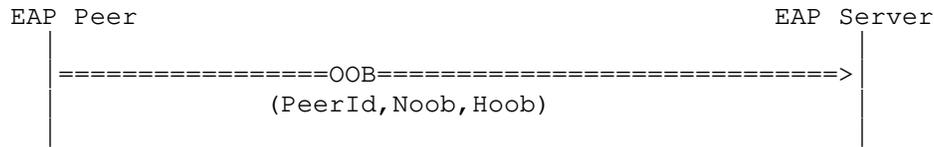


Figure 4: OOB Step, from peer to EAP server

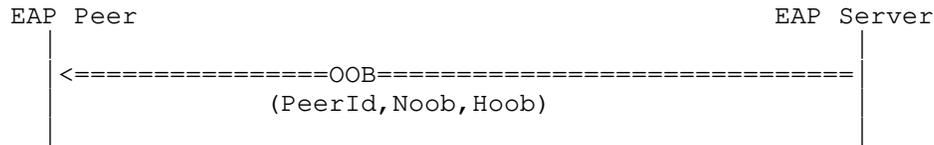


Figure 5: OOB Step, from EAP server to peer

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

the OOB receiver notifies the user about redundant OOB messages, but it MAY 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 $\text{NoobInterval} = \text{NoobTimeout} / 2$, 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 dynamic QR code 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 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.

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 K_{ms} and K_{mp} as defined in Section 3.5 and the message authentication codes MAC_s and MAC_p 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 MAC_s and the server finds that it has received the correct value of MAC_p , 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 called 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.

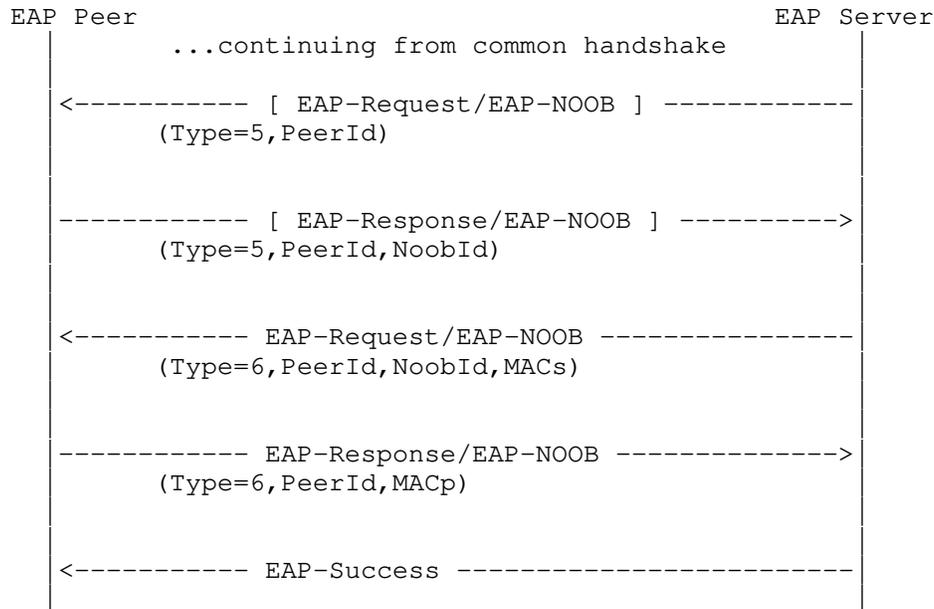


Figure 6: Completion Exchange

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 always ends 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 SHOULD 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 MUST 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

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 message authentication code. 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`.

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.

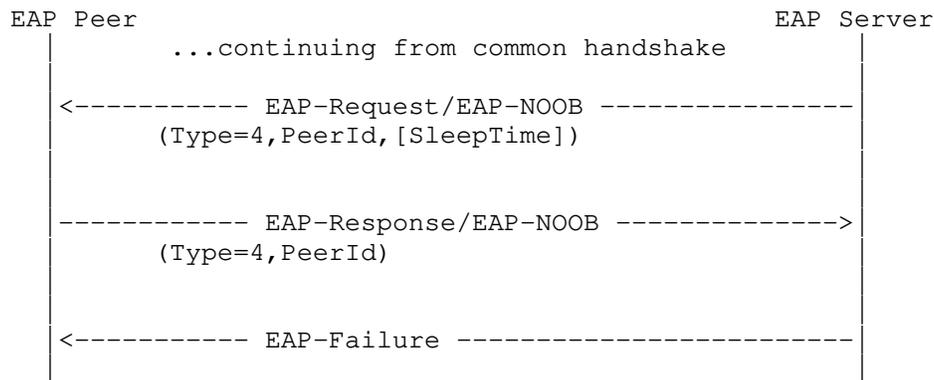


Figure 7: Waiting Exchange

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 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 ASCII string.

Another way to generate the identifiers is to choose a random 22-character alphanumeric ASCII 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. The peer MUST copy the PeerId byte-by-byte from the message where it was allocated, and the server MUST perform a byte-by-byte comparison between the received and the previously allocated PeerID. The peer 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 NAI for the peer is "noob@eap-noob.arpa". The peer implementation MAY allow the user or application to configure a different NAI, which overrides the default NAI. Furthermore, the server MAY assign a new NAI to the peer in the Initial Exchange or Reconnect Exchange, in the NewNAI field of request types 2 and 7, to override any previous NAI value. 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 NewNAI in the latest Initial Exchange, the peer MUST use the configured NAI, or if it does not exist, the default NAI. When the peer is in one of the states 1..2 and the server sent a NewNAI in the latest Initial Exchange, the peer MUST use this server-assigned NAI. When the peer moves to the Registered (4) state after the Completion Exchange, it writes to the persistent EAP-NOOB association the same NAI value that it used in the Completion Exchange. When the peer is in the Reconnecting (3) or Registered (4) state, it MUST use the NAI from its persistent EAP-NOOB association. When the server sends NewNAI in the Reconnect Exchange, the peer writes its value to the persistent EAP-NOOB association when it moves from the Reconnecting (3) state to the Registered (4) state. All the NAI values MUST follow the syntax specified in [RFC7542].

The purpose of the server-assigned NAI 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 realm part of the assigned NAI to determine peer-specific connection parameters, such as isolating the peer to a specific VLAN. The user or application configured NAI, on the other hand, 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 server-assigned NAI 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 NAI 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.

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.
NAI	Peer NAI 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 detailed 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

	<p>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.</p>
Dirs, Dirp	<p>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.</p>
Dir	<p>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.</p>
Ns, Np	<p>32-byte nonces for the Initial Exchange.</p>
ServerInfo	<p>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.</p>
PeerInfo	<p>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.</p>
SleepTime	<p>The number of seconds for which the peer MUST</p>

	<p>NOT start a new execution of the EAP-NOOB method with the authenticator, unless the peer receives the OOB message or the sending is triggered by an application-specific user action. 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.</p>
Noob	<p>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.</p>
Hoob	<p>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.</p>
NoobId	<p>16-byte identifier for the OOB message, computed with a one-way function from the nonce Noob in the message.</p>
MACs, MACp	<p>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.</p>
Ns2, Np2	<p>32-byte Nonces for the Reconnect Exchange.</p>
KeyingMode	<p>Integer indicating the key derivation method. 0 in the Completion Exchange, and 1..3 in the Reconnect Exchange.</p>
PKs2, PKp2	<p>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.</p>
MACs2, MACp2	<p>Message authentication codes (HMAC) for mutual</p>

	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 the 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, Cryptosuitep, Dirp, [NewNAI], PeerInfo, 0, PKs, Ns, PKp, Np, Noob) .
```

```
NoobId = H("NoobId", Noob) .
```

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

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

```
MACs2 = HMAC(Kms2; 2, Vers, Verp, PeerId, Cryptosuites, "", [ServerInfo], Cryptosuitep, "", [NewNAI], [PeerInfo], KeyingMode, [PKs2], Ns2, [PKp2], Np2, "")
```

```
MACp2 = HMAC(Kmp2; 1, Vers, Verp, PeerId, Cryptosuites, "", [ServerInfo], Cryptosuitep, "", [NewNAI], [PeerInfo], KeyingMode, [PKs2], Ns2, [PKp2], Np2, "")
```

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). NewNAI 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 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 ASCII characters)
Verp	Negotiated protocol version	integer
Cryptosuitep	Negotiated cryptosuite	integer
CryptosuitepPrev (at peer only)	Previous cryptosuite	integer
NAI	NAI assigned by server, configured by user, or the default NAI "noob@eap-noob.arpa"	UTF-8 string
Kz	Persistent key material	32 bytes
KzPrev (at peer only)	Previous Kz value	32 bytes

Table 2: Persistent EAP-NOOB association

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 CryptosuitepPrev from

the persistent EAP-NOOB association are only used to support the negotiation as described above; all actual cryptographic operations use the newly 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 value 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 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 `NewNAI`, `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 `NewNAI` was sent, each side updates `NAI` in the persistent EAP-NOOB association when moving to the Registered (4) state.

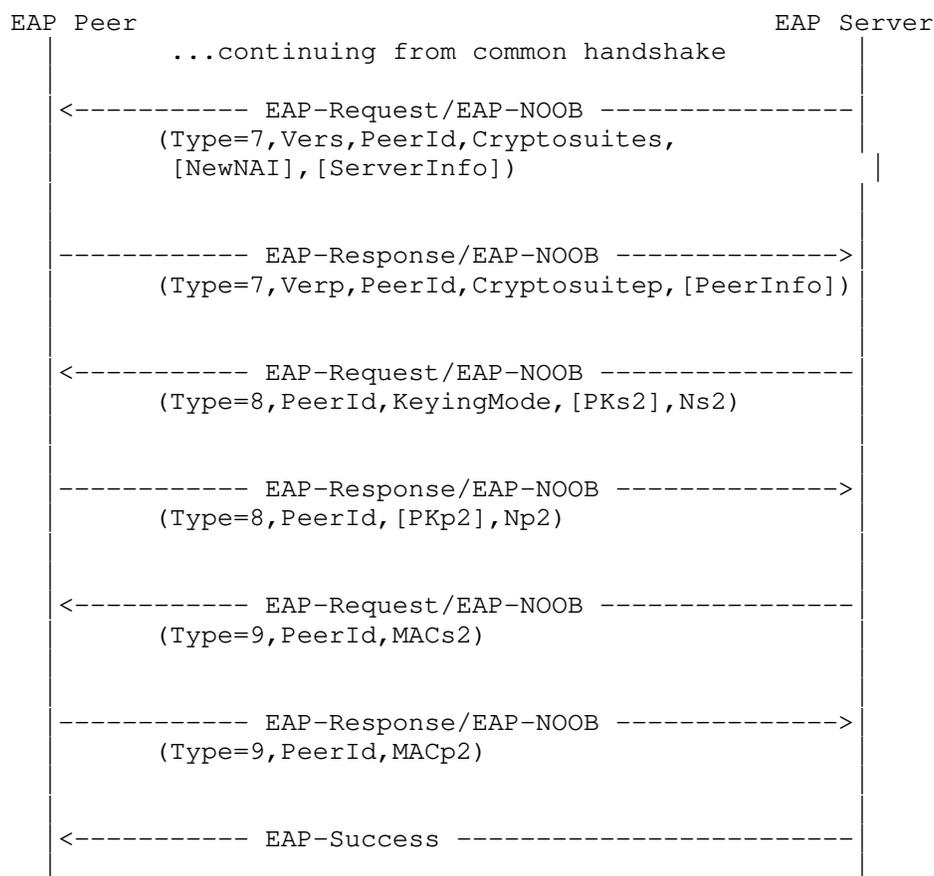


Figure 8: Reconnect Exchange

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

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 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 one-step key derivation function (KDF) defined in section 5.8.2.1 of the NIST specification. The auxiliary function H is a hash function and it 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 (`FixedInfo`), for which we use the concatenation format defined in section 5.8.2.1.1 of the NIST specification [NIST-DH]. `FixedInfo` 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 SuppPrivInfo	(not allowed) Noob	16
1 Reconnect, rekeying without ECDHE	Z	Kz	32
	AlgorithmId	"EAP-NOOB"	8
	PartyUInfo	Np2	32
	PartyVInfo	Ns2	32
	SuppPubInfo SuppPrivInfo	(not allowed) (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 SuppPrivInfo	(not allowed) 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 Completion	0..63	MSK	64
	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
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 a 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.

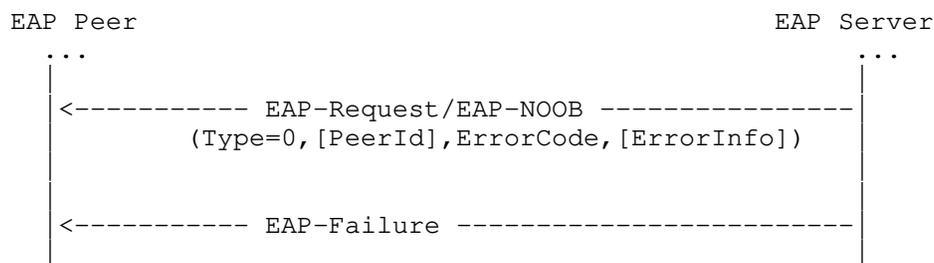


Figure 9: Error notification from server to peer

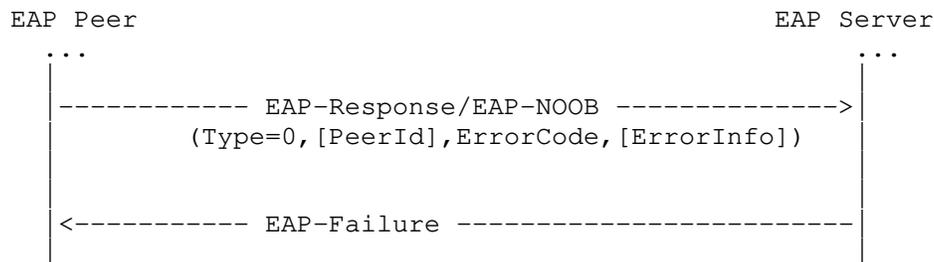


Figure 10: Error notification from peer to server

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 an 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. This range is reserved for "Private Use" and "Experimental Use", both locally and on the open Internet.

4.4. Domain name reservation considerations

The special-use domain "eap-noob.arpa" should be registered in the .arpa registry (<<https://www.iana.org/domains/arpa>>). The considerations required by [RFC6761] for registering this special-use domain name are the following:

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

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

5. Implementation Status

Note to RFC Editor: Please remove this entire section and the reference to RFC7942 before publication.

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

- o Responsible Organization: Aalto University
- o Location: <<https://github.com/tuomaura/eap-noob>>
- o 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.
- o Level of Maturity: Alpha
- o Version compatibility: Version 08 of the individual draft implemented
- o Licensing: BSD

- o Contact Information: Tuomas Aura, tuomas.aura@aalto.fi

5.2. Implementation on Contiki

- o Responsible Organization: University of Murcia and Aalto University
- o Location: <<https://github.com/eduingles/coap-eap-noob>>
- o 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.
- o Level of Maturity: Alpha
- o Version compatibility: Version 06 of the draft implemented
- o Licensing: BSD
- o Contact Information: Eduardo Ingles, eduardo.ingles@um.es

5.3. Implementation with wpa_supplicant and hostapd

- o Responsible Organization: Ericsson
- o Location: <<https://github.com/Vogeltak/hostap>>
- o Coverage: This implementation is the most up-to-date. The implementation only provides a minimal API interface for transferring OOB messages.
- o Level of Maturity: Alpha
- o Version compatibility: Version 02 of the working group adopted draft is implemented
- o Licensing: BSD

5.4. Protocol modeling

The current EAP-NOOB specification has been modeled with the mCRL2 formal specification language [mcr12]. The model <<https://github.com/tuomaura/eap-noob/tree/master/protocolmodel/mcr12>> 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 disclosure of the OOB message, and such applications benefit from the second authentication feature.

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 during fast reconnect 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. A certificate or an attestation token (e.g., [I-D.tschofenig-tls-cwt] and [I-D.ietf-rats-eat]) 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.

After the device registration, an attacker could clone the device identity by copying the keys from the persistent EAP-NOOB association into another device. The attacker can be an outsider who gains access to the keys or the device owner who wants to have two devices matching the same registration. The cloning threats can be mitigated by creating the cryptographic keys and storing the persistent EAP-NOOB association on the peer device in a secure hardware component such as a trusted execution environment (TEE). Furthermore, remote attestation on the application level could provide assurance to the server that the device has not been cloned.

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 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 on the application level, the PeerInfo is not authenticated information and should not be relied on.

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 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, which may take place while roaming, the optional PeerInfo SHOULD be omitted unless some critical data has changed and it cannot be updated on the application layer.

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.

To enable stronger identity protection in later versions of EAP-NOOB, the optional server-assigned NAI (NewNAI) SHOULD have a constant username part. The RECOMMENDED username is "noob". The server MAY, however, send a different username in NewNAI to avoid username collisions within the realm or to conform to its local policy on usernames.

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 K_z 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 K_z . EAP-NOOB protects against such attacks by allowing cryptosuite upgrades in the Reconnect Exchange and by updating the shared key material K_z 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 K_z 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 the 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

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. Levkowitz, 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.
- [I-D.ietf-rats-eat] Mandyam, G., Lundblade, L., Ballesteros, M., and J. O'Donoghue, "The Entity Attestation Token (EAT)", draft-ietf-rats-eat-06 (work in progress), December 2020.
- [I-D.tschofenig-tls-cwt] Tschofenig, H. and M. Brossard, "Using CBOR Web Tokens (CWTs) in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", draft-tschofenig-tls-cwt-02 (work in progress), July 2020.
- [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>>.
- [RFC5903] Fu, D. and J. Solinas, "Elliptic Curve Groups modulo a Prime (ECP Groups) for IKE and IKEv2", RFC 5903, DOI 10.17487/RFC5903, June 2010, <<https://www.rfc-editor.org/info/rfc5903>>.
- [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 3	Initial Exchange -	both 1 (0 on error) no change, notify user
server state: Waiting for OOB (1)		
0 1 2 3	Initial Exchange Waiting Exchange Completion Exchange -	both 1 (0 on error) both 1 (no change on error) both 4 (A) no change, notify user
server state: OOB Received (2)		
0 1 2 3	Initial Exchange Completion Exchange Completion Exchange -	both 1 (0 on error) both 4 (B) both 4 (A) no change, notify user
server state: Reconnecting (3) or Registered (4)		
0..2 3	- Reconnect Exchange	no change, notify user 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	Mobility/timeout/network failure	0
3..4	Mobility/timeout/network failure	3
3..4	Rekeying request	3
0..4	User resets association	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.
ServerInfoType	The value of the Type field and the other required fields in ServerInfo
PeerInfoType	The value of the Type field and the other required fields 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. Further specifications may specify application-specific ServerInfo and PeerInfo contents.

Data field	Description
Type	Type-tag string that can be used by the peer as a hint for how to interpret the ServerInfo contents.
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 IEEE 802.11 wireless network identifier (SSID) strings used for roaming support, as suggested in Appendix D. JSON array of ASCII encoded SSID strings.
Base64SSIDList	List of IEEE 802.11 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.

Data field	Description
Type	Type-tag string that can be used by the server as a hint for how to interpret the PeerInfo contents.
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 be in upper or lower case and MAY include additional colon ':' or dash '-' characters that MUST be ignored by the server.
SSID	IEEE 802.11 network SSID for channel binding. The SSID is a ASCII string.
Base64SSID	IEEE 802.11 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] for channel binding. The string MAY be in upper or lower case and MAY include additional colon ':' or dash '-' characters that MUST be ignored by the server.

Table 12: Suggested PeerInfo data fields

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 allowing the server to assign a NAI

to the peer. After the NAI has been assigned, it 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 NAI of the home network. This is the user or application configured NAI mentioned in Section 3.3.1. In that case, the EAP-NOOB association can be created while roaming. The configured NAI 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 field 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.

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 dynamic 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, so that there is a secure connection to the server and the man-in-the-middle attacker cannot read or modify the OOB message.

The ServerInfo in this case includes a field called ServerUrl of the following format with RECOMMENDED length of 60 characters or less:

```
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 ASCII string. The peer base64url encodes, without padding, the 16-byte values Noob and Hoob into 22-character

ASCII 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://aaa.example.com/eapnoob?P=mcm5BSCDZ45cYPlAr1ghNw&N=rMinS0-F4E  
fCU8D9ljxX_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.arpa

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

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

EAP request (type 2):
{"Type":2,"Vers":[1],"PeerId":"07KRU6OgqX0HIeRFldnbSW","NewNAI":"noob@example.org","Cryptosuites":[1,2],"Dirs":3,"ServerInfo":{"Type":"url_wifi","Name":"Example","Url":"https://noob.example.org/sendOOB"}}

EAP response (type 2):
{"Type":2,"Verp":1,"PeerId":"07KRU6OgqX0HIeRFldnbSW","Cryptosuitep":1,"Dirp":2,"PeerInfo":{"Type":"wifi","Make":"Acme","Serial":"DU-9999","SSID":"Noob1","BSSID":"6c:19:8f:83:c2:80"}}

EAP request (type 3):
{"Type":3,"PeerId":"07KRU6OgqX0HIeRFldnbSW","PKs":{"kty":"EC","crv":"X25519","x":"hSDwCYkwp1R0i33ctD73Wg2_Og0mOBr066SpjqqbTmo"},"Ns":"PYO7NVd9Af3BxEri1MI6hL8Ck49YxwCjSRPq1C1SPbw","SleepTime":60}

EAP response (type 3):

```
 {"Type":3,"PeerId":"07KRU6OgqX0HIeRFldnbSW","PKp":{"kty":"EC","crv":
 "X25519","x":"3p7bfXt9wbTTW2HC7OQ1Nz-DQ8hbeGdNrfx-FG-
 IK08"},"Np":"HIvB6g0n2btpxEcU7YXnWB-451ED6L6veQQd6ugiPFU" }
```

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

```
 Identity response:
   noob@example.org
```

```
 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=unFfhYbHTzcDtV
   uMs07Bmw
```

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

```
 Identity response:
   noob@example.org
```

```
 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":"caIpiJr3xjjfg--w1s9CVgDiDrIHrCJQyZ_NWsRO9HQ" }
```

```
EAP response (type 6):
  {"Type":6,"PeerId":"07KRU6OgqX0HIeRFldnbSW","MACp":"5ouhafoqzbNY4S
  NiPlGckXynS5zx7sulTMQnlmrPXws"}
```

```
===== Reconnect Exchange (KeyingMode = 1) =====
```

```
Identity response:
  noob@example.org
```

```
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","Cryptosuites":[1,2]}
```

```
EAP response (type 7):
  {"Type":7,"Verp":1,"PeerId":"07KRU6OgqX0HIeRFldnbSW","Cryptosuitep":2}
```

```
EAP request (type 8):
  {"Type":8,"PeerId":"07KRU6OgqX0HIeRFldnbSW","KeyingMode":1,"Ns2":"
  RDLahHBlIgmL_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":"kQpDAb8W5set6
  juShu10Kqa_qZ12NDTAq6S6ca_RzXI"}
```

```
EAP response (type 9):
  {"Type":9,"PeerId":"07KRU6OgqX0HIeRFldnbSW","MACp2":"13-
  f_WAlQuriP-_OaHA9IY6gWYUsM5Ngz3hk83PLgBc"}
```

```
===== Reconnect Exchange (KeyingMode = 2) =====
```

```
Identity response:
  noob@example.org
```

```
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","Cryptosuites":[1,2]}

EAP response (type 7):
  {"Type":7,"Verp":1,"PeerId":"07KRU6OgqX0HIeRFldnbSW","Cryptosuitep":1}

EAP request (type 8):
  {"Type":8,"PeerId":"07KRU6OgqX0HIeRFldnbSW","KeyingMode":2,"PKs2":
  {"kty":"EC","crv":"X25519","x":"hSDwCYkwp1R0i33ctD73Wg2_Og0mOBr066
  SpjqqbTmo"},"Ns2":"RDLahHBlIgmL_F_xcynrHurLPkCsrp3G3B_S82WUF4"}

EAP response (type 8):
  {"Type":8,"PeerId":"07KRU6OgqX0HIeRFldnbSW","PKp2":{"kty":"EC","crv":
  "X25519","x":"3p7bfXt9wbTTW2HC7OQ1Nz-DQ8hbeGdNrfx-FG-IK08"},"Np2":
  "jN0_V4P0JoTqwI9VHHQKd9ozUh7tQdc9ABd-j6oTy_4"}

EAP request (type 9):
  {"Type":9,"PeerId":"07KRU6OgqX0HIeRFldnbSW","MACs2":"cjkfWIXhAnqBC
  0XbWgpbL3FFQ6KJBh2_E7dMig8QyoY"}

EAP response (type 9):
  {"Type":9,"PeerId":"07KRU6OgqX0HIeRFldnbSW","MACp2":"H-7SS9fJfAyPu
  nOq4MYCGYLRwN_xxklgf4WuqyCJ84g"}
```

===== Reconnect Exchange (KeyingMode = 3) =====

These message examples show the case where the peer upgrades to Cryptosuite 2 during the Reconnect Exchange. The messages are generated with NIST P-256 ECDHE test vectors specified in section 8.1 of [RFC5903] (server=responder, peer=initiator).

```
Identity response:
  noob@example.org
```

```
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","Cryptosuites":[1,2]}
```

```

EAP response (type 7):
  {"Type":7,"Verp":1,"PeerId":"07KRU6OgqX0HIeRFldnbSW","Cryptosuitep":2}

EAP request (type 8):
  {"Type":8,"PeerId":"07KRU6OgqX0HIeRFldnbSW","KeyingMode":3,"PKs2":
  {"kty":"EC","crv":"P-256","x":"2tC2U5QiHPmwUeH-yIeH0Jjf5jf8kLnv1F0
  MN3JYEYA","y":"UnGgRhzbglLWHxxFb6PlmrH0WzOsz19Y0J4Fd7iZC7M"},"Ns2":
  "RDLahHBlIgmL_F_xcynrHurLPkCsrp3G3B_S82WUF4"}

EAP response (type 8):
  {"Type":8,"PeerId":"07KRU6OgqX0HIeRFldnbSW","PKp2":{"kty":"EC","cr
  v":"P-256","x":"0S37UonI1PgSCLcCcDmMNCKWlwoLzLdMc2_HVUSUv2M","y":"
  VvzyjZswj6BV4VME8WNaqwj8Eatow-DU-
  dPMwOYcqs"},"Np2":"jN0_V4P0JoTqwI9VHHQKd9ozUh7tQdc9ABd-j6oTy_4"}

EAP request (type 9):
  {"Type":9,"PeerId":"07KRU6OgqX0HIeRFldnbSW","MACs2":"j94I1oGeUfVc7
  wrUHjPm_babIQUTUp5esSeAt2LhYC4"}

EAP response (type 9):
  {"Type":9,"PeerId":"07KRU6OgqX0HIeRFldnbSW","MACp2":"-aIFkKDNyCC8n
  S6_uEv0ZePLX2AQ_ZQOLYS5wqe7dAc"}

```

Appendix G. Version history

- o Version 01:
 - * Fixed Reconnection Exchange.
 - * URL examples.
 - * Message examples.
 - * Improved state transition (event) tables.
- o 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.
- o 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.
- o Version 07:
 - * Updated example messages.
 - * Added pointers to new implementation in Contiki.
- o Version 08:
 - * Editorial improvements and corrections.
- o WG Version 00:
 - * Editorial improvements and corrections.

- * Updated reference [NIST-DH] to Revision 3 (2018).
- o WG Version 01:
 - * Add NIST P-256 as Cryptosuite 2.
 - * Renumber message types.
 - * Very minor editorial fixes.
- o WG Version 02:
 - * Updated message examples with all KeyingModes.
 - * Many editorial fixes and other updates based on the IoT directorate review of Dave Thaler.
 - * Text on cloning attacks based on review by Hannes Tschofenig.
- o WG Version 03:
 - * Changed server-assigned Realm to server-assigned NAI to avoid username collisions. This issue was identified in a review by Alan Dekok.
 - * Minor editorial improvements.

Appendix H. Acknowledgments

Max Crone, Shiva Prasad TP, and Raghavendra MS implemented parts of this protocol with wpa_supplicant and hostapd. Eduardo Ingles (RFC editor: please add an accented e in Ingles) and Dan Garcia-Carrillo were involved in the implementation of this protocol on Contiki. Their inputs helped us in improving the specification.

The authors would 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, Alan Dekok, Darshak Thakore, Stefan Winter, Hannes Tschofenig, and Daniel Migault for their comments and reviews.

We would also like to express our sincere gratitude to Dave Thaler for his thorough review of the document.

Authors' Addresses

Tuomas Aura
Aalto University
Aalto 00076
Finland

EMail: tuomas.aura@aalto.fi

Mohit Sethi
Ericsson
Jorvas 02420
Finland

EMail: mohit@piuha.net

Aleksi Peltonen
Aalto University
Aalto 00076
Finland

EMail: aleksi.peltonen@aalto.fi

EMU Working Group
Internet-Draft
Intended status: Experimental
Expires: May 6, 2021

E. Ingles
University of Murcia
D. Garcia-Carrillo
University of Oviedo
R. Marin-Lopez
University of Murcia
November 2, 2020

EAP method based on EDHOC Authentication
draft-ingles-eap-edhoc-01

Abstract

This document describes a proposal of an EAP method based on the EDHOC authentication protocol.

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 May 6, 2021.

Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Requirements Language	2
2. Protocol Overview	2
2.1. The EAP-EDHOC Conversation	2
2.1.1. Transport and Message Correlation	4
2.1.2. Identity	5
3. Identity Verification	5
4. Key Hierarchy	5
5. IANA considerations	6
6. Security Considerations	6
7. Acknowledgements	6
8. Normative References	7
Authors' Addresses	7

1. Introduction

EDHOC [I-D.selander-lake-edhoc] is a new protocol for authentication and key derivation that has been proposed as an alternative in IoT to provide a secure exchange in an end-to-end fashion. This key material can be further used to run other protocols such as OSCORE, as well as providing key material to any other protocol that needs pre-shared key material to secure the communications. Provides authentication and key material generation, which are basic pillars to the design of an EAP method. And indeed the most important thing is that it is lightweight and designed for IoT. In addition, the EDHOC implementation that exists on the device can be reused to establish OSCORE Security Associations (SAs) for the authentication process. EAP is a protocol that allows to implement different authentication mechanisms, provides a framework for key management and has integration with AAA infrastructures. For these reasons, this new EAP method will allow the different applications and use cases to take advantage of EAP.

1.1. Requirements Language

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 RFC 2119 [RFC2119].

2. Protocol Overview

2.1. The EAP-EDHOC Conversation

The exchange of messages befalls between two entities that EDHOC identifies as Initiator (I) and Responder (R). In this EAP method,

we establish equivalence with those terms. On the one hand, EAP peer acts as Initiator while EAP Server takes on the role of Responder.

The EAP-EDHOC conversation typically starts with the negotiation of EAP by the EAP authenticator and the EAP peer. The EAP authenticator sends an EAP-Request/Identity packet to the EAP peer, to which the EAP peer answers with an EAP-Response/Identity. This last messages contains the peer's user-Id.

From this point on, the authenticator MAY act as a forwarder of the EAP messages between the EAP peer and the server, if the pass-through mode is used, receiving the EAP packets from the peer, encapsulating them for transmission to the EAP server that will act as Authentication Server (AS).

Once the EAP server receives the peer's Identity, it MUST respond with an empty EAP-EDHOC/Start message, which is an EAP-Request packet with EAP-Type=EAP-EDHOC and no data. Initiator must initiate the EDHOC conversation. Hence, EAP Server sends this message to indicate that it can start the authentication process. The EAP-EDHOC conversation will then begin, with the peer sending an EAP-Response packet with EAP-Type=EAP-EDHOC. The data field of that packet will encapsulate the "EDHOC Message 1".

The EAP server will then respond with an EAP-Request packet with EAP-Type=EAP-EDHOC. The data field of this packet will encapsulate "EDHOC Message 2" message. To this message, the EAP peer will send the and EAP-Response message containing the "EDHOC Message 3" message.

In the case where the EDHOC mutual authentication is successful, the conversation will appear as follows:

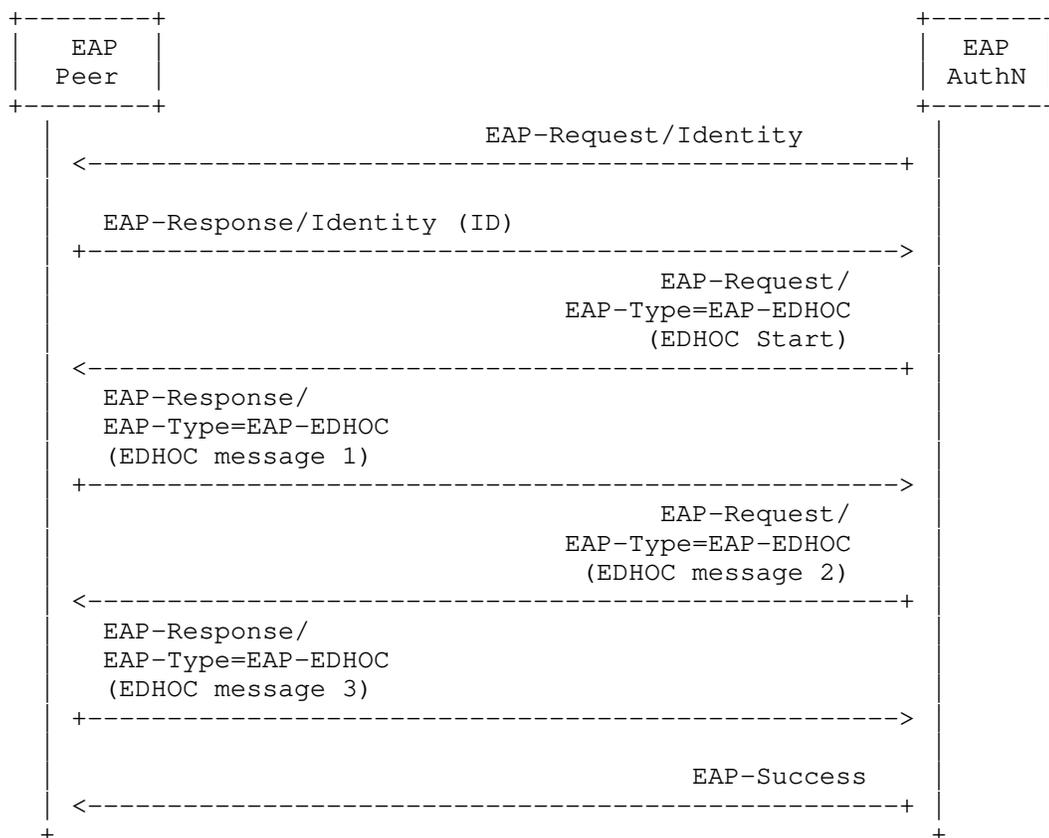


Figure 1: Overview EDHOC exchange

2.1.1. Transport and Message Correlation

One of the defining characteristics of EAP is its lock-step procedure. The EAP protocol manages the exchange of messages guaranteeing the order of transmission. In the same way, it manages retransmissions and the detection of duplicate messages. Therefore, EAP ensures the message correlation mechanism in the different EAP layers.

Given the above, EDHOC does not need to use its internal mechanism for correlating messages. Then, the value for METHOD_CORR variable must satisfy the formula:

$$\text{METHOD_CORR} = 4 * \text{method} + \text{corr}$$

Where:

method = EDHOC Method Type defined in Section 8.2 of EDHOC
[I-D.selander-lake-edhoc]

corr =

2.1.2. Identity

It is RECOMMENDED to use NAIs in the Identity Response as identities.

3. Identity Verification

The identity provided in the EAP-Response/Identity is not authenticated by EAP-EDHOC, hence SHALL NOT be used for authorization or accounting purposes. The authenticator and the EAP server MAY examine the identity presented in EAP-Response/Identity for routing and EAP method selection.

4. Key Hierarchy

EDHOC uses HKDF RFC 5869 [RFC5869] to derive keys. HKDF-Extract is used for deriving fixed-length uniformly pseudorandom keys (PRK) from ECDH shared secrets. HKDF-Expand is used for deriving additional output keying material (OKM) from the PRKs.

The derivation proceeds as follows:

PRK = HKDF-Extract(salt, IKM)

Where:

HKDF-Extract = RFC5869 HKDF function

salt = The empty byte string

IKM (input keying material) = The ECDH shared secret

Figure 2 illustrates the EDHOC Key Hierarchy.

In EAP-EDHOC, the MSK, EMSK, and Initialization Vector (IV) are derived from the PRK via a hash function. This ensures that the EDHOC PRK cannot be derived from the MSK, EMSK, or IV unless the hash function is defeated. Since the MSK and EMSK are derived from the EDHOC PRK, if the EDHOC PRK is compromised then the MSK and EMSK are also compromised.

EAP-EDHOC derives exported keying material and parameters as follows:

```

Type-Code = 0XFF
Key_Material = HKDF-Expand(EDHOC PRK, "EAP-EDHOC encryption", 128)
MSK = Key_Material(0,63)
EMSK = Key_Material(64,127)
IV = HKDF-Expand(EDHOC PRK, "EAP-EDHOC IV", 64)
Session-Id = Type-Code || Method-Id
Method-Id = HKDF-Expand(EDHOC PRK, "EAP_EDHOC_Method-Id", 64)
    
```

Where:

Key_Material(S,F) = Octets S through F inclusive of the key material.

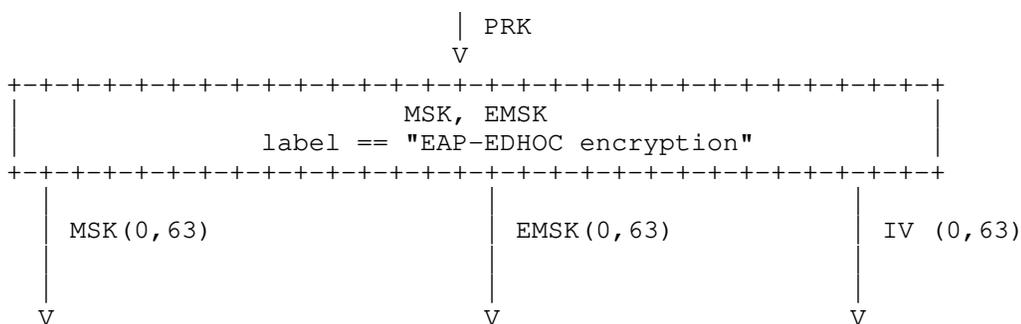


Figure 2: EAP-EDHOC Key derivation

5. IANA considerations

TBD.

6. Security Considerations

TBD.

7. Acknowledgements

This work is possible due the EU Project IoTcrawler under grant agreement n.779852 and the EU Project INSPIRE-5Gplus under grant agreement n.871808

8. Normative References

- [I-D.selander-lake-edhoc]
Selander, G., Mattsson, J., and F. Palombini, "Ephemeral Diffie-Hellman Over COSE (EDHOC)", draft-selander-lake-edhoc-01 (work in progress), March 2020.
- [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>>.
- [RFC2548] Zorn, G., "Microsoft Vendor-specific RADIUS Attributes", RFC 2548, DOI 10.17487/RFC2548, March 1999, <<https://www.rfc-editor.org/info/rfc2548>>.
- [RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <<https://www.rfc-editor.org/info/rfc5869>>.

Authors' Addresses

Eduardo Ingles-Sanchez
University of Murcia
Campus de Espinardo S/N, Faculty of Computer Science
Murcia 30100
Spain

Email: eduardo.ingles@um.es

Dan Garcia-Carrillo
University of Oviedo
Campus de Gijon, S/N, Escuela Politecnica de Ingenieria de Gijon
Gijon, Asturias 33203
Spain

Email: garciadan@uniovi.es

Rafael Marin-Lopez
University of Murcia
Campus de Espinardo S/N, Faculty of Computer Science
Murcia 30100
Spain

Phone: +34 868 88 85 01
Email: rafa@um.es