Internet Engineering Task Force (IETF)
Request for Comments: 8037
Category: Standards Track
ISSN: 2070-1721

I. Liusvaara
Independent

January 2017

CFRG Elliptic Curve Diffie-Hellman (ECDH) and Signatures in JSON Object Signing and Encryption (JOSE)

Abstract

This document defines how to use the Diffie-Hellman algorithms "X25519" and "X448" as well as the signature algorithms "Ed25519" and "Ed448" from the IRTF CFRG elliptic curves work in JSON Object Signing and Encryption (JOSE).

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc8037.

Table of Contents

1. Introduction 2
1.1. Terminology 3
2. Key Type "OKP" 3
3. Algorithms 4
3.1. Signatures 4
3.1.1. Signing 4
3.1.2. Verification 4
3.2. ECDH-ES 4
3.2.1. Performing the ECDH Operation 5
4. Security Considerations 5
5. IANA Considerations 6
6. References 8
6.1. Normative References 8
6.2. Informative References 8
Appendix A. Examples 9
A.1. Ed25519 Private Key 9
A.2. Ed25519 Public Key 9
A.3. JWK Thumbprint Canonicalization 9
A.4. Ed25519 Signing 10
A.5. Ed25519 Validation 11
A.6. ECDH-ES with X25519 11
A.7. ECDH-ES with X448 12
Acknowledgements 14
Author's Address 14

1. Introduction

The Internet Research Task Force (IRTF) Crypto Forum Research Group (CFRG) selected new Diffie-Hellman algorithms ("X25519" and "X448"; [RFC7748]) and signature algorithms ("Ed25519" and "Ed448"; [RFC8032]) for asymmetric key cryptography. This document defines how to use those algorithms in JOSE in an interoperable manner.

This document defines the conventions to use in the context of [RFC7515], [RFC7516], and [RFC7517].

While the CFRG also defined two pairs of isogenous elliptic curves that underlie these algorithms, these curves are not directly exposed, as the algorithms laid on top are sufficient for the purposes of JOSE and are much easier to use.

All inputs to and outputs from the Elliptic Curve Diffie-Hellman (ECDH) and signature functions are defined to be octet strings, with the exception of outputs of verification functions, which are booleans.

1.1. 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].

"JWS Signing Input" and "JWS Signature" are defined by [RFC7515].

"Key Agreement with Elliptic Curve Diffie-Hellman Ephemeral Static" is defined by Section 4.6 of [RFC7518].

The JOSE key format ("JSON Web Key (JWK)") is defined by [RFC7517] and thumbprints for it ("JSON Web Key (JWK) Thumbprint") in [RFC7638].

2. Key Type "OKP"

A new key type (kty) value "OKP" (Octet Key Pair) is defined for public key algorithms that use octet strings as private and public keys. It has the following parameters:

- The parameter "kty" MUST be "OKP".
- The parameter "crv" MUST be present and contain the subtype of the key (from the "JSON Web Elliptic Curve" registry).
- The parameter "x" MUST be present and contain the public key encoded using the base64url [RFC4648] encoding.
- The parameter "d" MUST be present for private keys and contain the private key encoded using the base64url encoding. This parameter MUST NOT be present for public keys.

Note: Do not assume that there is an underlying elliptic curve, despite the existence of the "crv" and "x" parameters. (For instance, this key type could be extended to represent Diffie-Hellman (DH) algorithms based on hyperelliptic surfaces.)

When calculating JWK Thumbprints [RFC7638], the three public key fields are included in the hash input in lexicographic order: "crv", "kty", and "x".

3. Algorithms

3.1. Signatures

For the purpose of using the Edwards-curve Digital Signature Algorithm (EdDSA) for signing data using "JSON Web Signature (JWS)" [RFC7515], algorithm "EdDSA" is defined here, to be applied as the value of the "alg" parameter.

The following key subtypes are defined here for use with EdDSA:

"crv"	EdDSA Variant
Ed25519	Ed25519
Ed448	Ed448

The key type used with these keys is "OKP" and the algorithm used for signing is "EdDSA". These subtypes MUST NOT be used for Elliptic Curve Diffie-Hellman Ephemeral Static (ECDH-ES).

The EdDSA variant used is determined by the subtype of the key (Ed25519 for "Ed25519" and Ed448 for "Ed448").

3.1.1. Signing

Signing for these is performed by applying the signing algorithm defined in [RFC8032] to the private key (as private key), public key (as public key), and the JWS Signing Input (as message). The resulting signature is the JWS Signature. All inputs and outputs are octet strings.

3.1.2. Verification

Verification is performed by applying the verification algorithm defined in [RFC8032] to the public key (as public key), the JWS Signing Input (as message), and the JWS Signature (as signature). All inputs are octet strings. If the algorithm accepts, the signature is valid; otherwise, the signature is invalid.

3.2. ECDH-ES

The following key subtypes are defined here for purpose of "Key Agreement with Elliptic Curve Diffie-Hellman Ephemeral Static" (ECDH-ES):

"crv"	ECDH Function Applied
X25519	X25519
X448	X448

The key type used with these keys is "OKP". These subtypes MUST NOT be used for signing.

Section 4.6 of [RFC7518] defines the ECDH-ES algorithms "ECDH-ES+A128KW", "ECDH-ES+A192KW", "ECDH-ES+A256KW", and "ECDH-ES".

3.2.1. Performing the ECDH Operation

The "x" parameter of the "epk" field is set as follows:

Apply the appropriate ECDH function to the ephemeral private key (as scalar input) and the standard base point (as u-coordinate input). The base64url encoding of the output is the value for the "x" parameter of the "epk" field. All inputs and outputs are octet strings.

The Z value (raw key agreement output) for key agreement (to be used in subsequent Key Derivation Function (KDF) as per Section 4.6.2 of [RFC7518]) is determined as follows:

Apply the appropriate ECDH function to the ephemeral private key (as scalar input) and receiver public key (as u-coordinate input). The output is the Z value. All inputs and outputs are octet strings.

4. Security Considerations

Security considerations from [RFC7748] and [RFC8032] apply here.

Do not separate key material from information about what key subtype it is for. When using keys, check that the algorithm is compatible with the key subtype for the key. To do otherwise opens the system up to attacks via mixing up algorithms. It is particularly dangerous to mix up signature and Message Authentication Code (MAC) algorithms.

Although for Ed25519 and Ed448, the signature binds the key used for signing, do not assume this, as there are many signature algorithms that fail to make such a binding. If key-binding is desired, include the key used for signing either inside the JWS protected header or the data to sign.

If key generation or batch signature verification is performed, a well-seeded cryptographic random number generator is REQUIRED. Signing and non-batch signature verification are deterministic operations and do not need random numbers of any kind.

The JSON Web Algorithm (JWA) ECDH-ES KDF construction does not mix keys into the final shared secret. In key exchange, such mixing could be a bad mistake; whereas here either the receiver public key has to be chosen maliciously or the sender has to be malicious in order to cause problems. In either case, all security evaporates.

The nominal security strengths of X25519 and X448 are ~126 and ~223 bits. Therefore, using 256-bit symmetric encryption (especially key wrapping and encryption) with X448 is RECOMMENDED.

5. IANA Considerations

The following has been added to the "JSON Web Key Types" registry:

- "kty" Parameter Value: "OKP"
- Key Type Description: Octet string key pairs
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 2 of RFC 8037

The following has been added to the "JSON Web Key Parameters" registry:

- Parameter Name: "crv"
- Parameter Description: The subtype of key pair
- Parameter Information Class: Public
- Used with "kty" Value(s): "OKP"
- Change Controller: IESG
- Specification Document(s): Section 2 of RFC 8037

- Parameter Name: "d"
- Parameter Description: The private key
- Parameter Information Class: Private
- Used with "kty" Value(s): "OKP"
- Change Controller: IESG
- Specification Document(s): Section 2 of RFC 8037

- Parameter Name: "x"
- Parameter Description: The public key
- Parameter Information Class: Public
- Used with "kty" Value(s): "OKP"
- Change Controller: IESG
- Specification Document(s): Section 2 of RFC 8037

The following has been added to the "JSON Web Signature and Encryption Algorithms" registry:

- Algorithm Name: "EdDSA"
- Algorithm Description: EdDSA signature algorithms
- Algorithm Usage Location(s): "alg"
- JOSE Implementation Requirements: Optional
- Change Controller: IESG

- Specification Document(s): Section 3.1 of RFC 8037
- Algorithm Analysis Documents(s): [RFC8032]

The following has been added to the "JSON Web Key Elliptic Curve" registry:

- Curve Name: "Ed25519"
- Curve Description: Ed25519 signature algorithm key pairs
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 3.1 of RFC 8037

- Curve Name: "Ed448"
- Curve Description: Ed448 signature algorithm key pairs
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 3.1 of RFC 8037

- Curve name: "X25519"
- Curve Description: X25519 function key pairs
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 3.2 of RFC 8037
- Analysis Documents(s): [RFC7748]

- Curve Name: "X448"
- Curve Description: X448 function key pairs
- JOSE Implementation Requirements: Optional
- Change Controller: IESG
- Specification Document(s): Section 3.2 of RFC 8037
- Analysis Documents(s): [RFC7748]

6. References

6.1. Normative References

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

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

[RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <http://www.rfc-editor.org/info/rfc7515>.

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

[RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <http://www.rfc-editor.org/info/rfc7518>.

[RFC7638] Jones, M. and N. Sakimura, "JSON Web Key (JWK) Thumbprint", RFC 7638, DOI 10.17487/RFC7638, September 2015, <http://www.rfc-editor.org/info/rfc7638>.

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

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <http://www.rfc-editor.org/info/rfc8032>.

6.2. Informative References

[RFC7516] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, DOI 10.17487/RFC7516, May 2015, <http://www.rfc-editor.org/info/rfc7516>.

Appendix A. Examples

To the extent possible, these examples use material taken from test vectors of [RFC7748] and [RFC8032].

A.1. Ed25519 Private Key

```
{"kty":"OKP","crv":"Ed25519",
"d":"nWGxne_9WmC6hEr0kuwsxERJxWl7MmkZcDusAxyuf2A",
"x":"11qYAYKxCrfVS_7TyWQHOg7hcvPapiMlrwIaaPcHURo"}
```

The hexadecimal dump of private key is:

```
9d 61 b1 9d ef fd 5a 60 ba 84 4a f4 92 ec 2c c4
44 49 c5 69 7b 32 69 19 70 3b ac 03 1c ae 7f 60
```

And of the public key is:

```
d7 5a 98 01 82 b1 0a b7 d5 4b fe d3 c9 64 07 3a
0e e1 72 f3 da a6 23 25 af 02 1a 68 f7 07 51 1a
```

A.2. Ed25519 Public Key

This is the public part of the previous private key (which just omits "d"):

```
{"kty":"OKP","crv":"Ed25519",
"x":"11qYAYKxCrfVS_7TyWQHOg7hcvPapiMlrwIaaPcHURo"}
```

A.3. JWK Thumbprint Canonicalization

The JWK Thumbprint canonicalization of the two examples above (with a linebreak inserted for formatting reasons) is:

```
{"crv":"Ed25519","kty":"OKP","x":"11qYAYKxCrfVS_7TyWQHOg7hcvPapiMlrwI
aaPcHURo"}
```

Which has the SHA-256 hash (in hexadecimal) of 90facafea9b1556698540f70c0117a22ea37bd5cf3ed3c47093c1707282b4b89, which results in the base64url encoded JWK Thumbprint representation of "kPrK_qmxVWaYVA9wwBF6Iuo3vVzz7TxHCTwXBygrS4k".

A.4. Ed25519 Signing

The JWS protected header is:

{"alg":"EdDSA"}

This has the base64url encoding of:

eyJhbGciOiJFZERTQSJ9

The payload is (text):

Example of Ed25519 signing

This has the base64url encoding of:

RXhhbXBsZSBvZiBFZDI1NTE5IHNpZ25pbmc

The JWS signing input is (a concatenation of base64url encoding of the (protected) header, a dot, and base64url encoding of the payload) is:

eyJhbGciOiJFZERTQSJ9.RXhhbXBsZSBvZiBFZDI1NTE5IHNpZ25pbmc

Applying the Ed25519 signing algorithm using the private key, public key, and the JWS signing input yields the signature (hex):

```
86 0c 98 d2 29 7f 30 60 a3 3f 42 73 96 72 d6 1b
53 cf 3a de fe d3 d3 c6 72 f3 20 dc 02 1b 41 1e
9d 59 b8 62 8d c3 51 e2 48 b8 8b 29 46 8e 0e 41
85 5b 0f b7 d8 3b b1 5b e9 02 bf cc b8 cd 0a 02
```

Converting this to base64url yields:

```
hgyY0il_MGCjP0JzlnLWG1PPOt7-09PGcvMg3AIbQR6dWbhijcNR4ki4iylGjg5BhVsPt
9g7sVvpAr_MuM0KAg
```

So the compact serialization of the JWS is (a concatenation of signing input, a dot, and base64url encoding of the signature):

```
eyJhbGciOiJFZERTQSJ9.RXhhbXBsZSBvZiBFZDI1NTE5IHNpZ25pbmc.hgyY0il_MGCj
P0JzlnLWG1PPOt7-09PGcvMg3AIbQR6dWbhijcNR4ki4iylGjg5BhVsPt9g7sVvpAr_Mu
M0KAg
```

A.5. Ed25519 Validation

The JWS from the example above is:

```
eyJhbGciOiJFZERTQSJ9.RXhhbXBsZSBvZiBFZDI1NTE5IHNpZ25pbmc.hgyY0il_MGCj
P0JzlnLWG1PPOt7-09PGcvMg3AIbQR6dWbhijcNR4ki4iylGjg5BhVsPt9g7sVvpAr_Mu
M0KAg
```

This has 2 dots in it, so it might be valid a JWS. Base64url decoding the protected header yields:

```
{"alg":"EdDSA"}
```

So this is an EdDSA signature. Now the key has: "kty":"OKP" and "crv":"Ed25519", so the signature is Ed25519 signature.

The signing input is the part before the second dot:

```
eyJhbGciOiJFZERTQSJ9.RXhhbXBsZSBvZiBFZDI1NTE5IHNpZ25pbmc
```

Applying the Ed25519 verification algorithm to the public key, JWS signing input, and the signature yields true. So the signature is valid. The message is the base64url decoding of the part between the dots:

Example of Ed25519 Signing

A.6. ECDH-ES with X25519

The public key to encrypt to is:

```
{"kty":"OKP","crv":"X25519","kid":"Bob",
"x":"3p7bfXt9wbTTW2HC7OQ1Nz-DQ8hbeGdNrfx-FG-IK08"}
```

The public key from the target key is (hex):

```
de 9e db 7d 7b 7d c1 b4 d3 5b 61 c2 ec e4 35 37
3f 83 43 c8 5b 78 67 4d ad fc 7e 14 6f 88 2b 4f
```

The ephemeral secret happens to be (hex):

```
77 07 6d 0a 73 18 a5 7d 3c 16 c1 72 51 b2 66 45
df 4c 2f 87 eb c0 99 2a b1 77 fb a5 1d b9 2c 2a
```

So the ephemeral public key is X25519(ephkey, G) (hex):

```
85 20 f0 09 89 30 a7 54 74 8b 7d dc b4 3e f7 5a
0d bf 3a 0d 26 38 1a f4 eb a4 a9 8e aa 9b 4e 6a
```

This is represented as the ephemeral public key value:

```
{"kty":"OKP","crv":"X25519",
"x":"hSDwCYkwp1R0i33ctD73Wg2_Og0mOBr066SpjqqbTmo"}
```

So the protected header could be, for example:

```
{"alg":"ECDH-ES+A128KW","epk":{"kty":"OKP","crv":"X25519",
"x":"hSDwCYkwp1R0i33ctD73Wg2_Og0mOBr066SpjqqbTmo"},
"enc":"A128GCM","kid":"Bob"}
```

And the sender computes the DH Z value as X25519(ephkey, recv_pub) (hex):

```
4a 5d 9d 5b a4 ce 2d e1 72 8e 3b f4 80 35 0f 25
e0 7e 21 c9 47 d1 9e 33 76 f0 9b 3c 1e 16 17 42
```

The receiver computes the DH Z value as X25519(seckey, ephkey_pub) (hex):

```
4a 5d 9d 5b a4 ce 2d e1 72 8e 3b f4 80 35 0f 25
e0 7e 21 c9 47 d1 9e 33 76 f0 9b 3c 1e 16 17 42
```

This is the same as the sender’s value (both sides run this through the KDF before using it as a direct encryption key or AES128-KW key).

A.7. ECDH-ES with X448

The public key to encrypt to (with a linebreak inserted for formatting reasons) is:

```
{"kty":"OKP","crv":"X448","kid":"Dave",
"x":"PreoKbDNIPW8_AtZm2_sz22kYnEHvbDU80W0MCfYuXL8PjT7QjKhPKcG3LV67D2
uB73BxnvzNgk"}
```

The public key from the target key is (hex):

```
3e b7 a8 29 b0 cd 20 f5 bc fc 0b 59 9b 6f ec cf
6d a4 62 71 07 bd b0 d4 f3 45 b4 30 27 d8 b9 72
fc 3e 34 fb 42 32 a1 3c a7 06 dc b5 7a ec 3d ae
07 bd c1 c6 7b f3 36 09
```

The ephemeral secret happens to be (hex):

```
9a 8f 49 25 d1 51 9f 57 75 cf 46 b0 4b 58 00 d4
ee 9e e8 ba e8 bc 55 65 d4 98 c2 8d d9 c9 ba f5
74 a9 41 97 44 89 73 91 00 63 82 a6 f1 27 ab 1d
9a c2 d8 c0 a5 98 72 6b
```

So the ephemeral public key is X448(ephkey, G) (hex):

```
9b 08 f7 cc 31 b7 e3 e6 7d 22 d5 ae a1 21 07 4a
27 3b d2 b8 3d e0 9c 63 fa a7 3d 2c 22 c5 d9 bb
c8 36 64 72 41 d9 53 d4 0c 5b 12 da 88 12 0d 53
17 7f 80 e5 32 c4 1f a0
```

This is packed into the ephemeral public key value (a linebreak inserted for formatting purposes):

```
{"kty":"OKP","crv":"X448",
"x":"mwj3zDG34-Z9ItWuoSEHSic70rg94Jxj-qc9LCLF2bvINmRyQdlT1AxbEtqIEg1
TF3-A5TLEH6A"}
```

So the protected header could be, for example (a linebreak inserted for formatting purposes):

```
{"alg":"ECDH-ES+A256KW","epk":{"kty":"OKP","crv":"X448",
"x":"mwj3zDG34-Z9ItWuoSEHSic70rg94Jxj-qc9LCLF2bvINmRyQdlT1AxbEtqIEg1
TF3-A5TLEH6A"},"enc":"A256GCM","kid":"Dave"}
```

And the sender computes the DH Z value as X448(ephkey,recv_pub) (hex):

```
07 ff f4 18 1a c6 cc 95 ec 1c 16 a9 4a 0f 74 d1
2d a2 32 ce 40 a7 75 52 28 1d 28 2b b6 0c 0b 56
fd 24 64 c3 35 54 39 36 52 1c 24 40 30 85 d5 9a
44 9a 50 37 51 4a 87 9d
```

The receiver computes the DH Z value as X448(seckey, ephkey_pub) (hex):

```
07 ff f4 18 1a c6 cc 95 ec 1c 16 a9 4a 0f 74 d1
2d a2 32 ce 40 a7 75 52 28 1d 28 2b b6 0c 0b 56
fd 24 64 c3 35 54 39 36 52 1c 24 40 30 85 d5 9a
44 9a 50 37 51 4a 87 9d
```

This is the same as the sender’s value (both sides run this through KDF before using it as the direct encryption key or AES256-KW key).

Acknowledgements

Thanks to Michael B. Jones for his comments on an initial draft of this document and editorial help.

Thanks to Matt Miller for some editorial help.

Author's Address

Ilari Liusvaara
Independent

Email: ilariliusvaara@welho.com