Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: December 13, 2015

Y. Nir
Check Point
S. Josefsson
SJD

June 11, 2015

Using Curve25519 for IKEv2 Key Agreement
draft-nir-ipsecme-curve25519-00

Abstract

This document describes the use of Curve25519 for ephemeral key exchange in the Internet Key Exchange (IKEv2) 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 http://datatracker.ietf.org/drafts/current/.

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

This Internet-Draft will expire on December 13, 2015.

Copyright Notice

Table of Contents

1. Introduction

[CFRG-Curves] specifies a new elliptic curve function for use in cryptographic applications. Curve25519 is a Diffie-Hellman function designed with performance and security in mind.

Almost ten years ago [RFC4753] specified the first elliptic curve Diffie-Hellman groups for the Internet Key Exchange protocol (IKEv2 - [RFC7296]). These were the so-called NIST curves. The state of the art has advanced since then. More modern curves allow faster implementations while making it much easier to write constant-time implementations free from side-channel attacks. This document defines such a curve for use in IKE. See [Curve25519] for details about the speed and security of this curve.

1.1. Conventions Used in This Document

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].

2. Curve25519

All cryptographic computations are done using the Curve25519 function defined in [CFRG-Curves]. In this document, this function is considered a black box that takes for input a (secret key, public key) pair and outputs a public key. Public keys are defined as strings of 32 octets. Secret keys are defined as 255-bit numbers such that high-order bit (bit 254) is set, and the three lowest-order bits are unset. In addition, a common public key, denoted by G, is shared by all users.

An ephemeral Diffie-Hellman key exchange using Curve25519 goes as follows: Each party picks a secret key d uniformly at random and computes the corresponding public key:

```
x_mine = Curve25519(d, G)
```

Parties exchange their public keys (see Section 3.1) and compute a shared secret:

```
SHARED_SECRET = Curve25519(d, x_peer).
```

This shared secret is used directly as the value denoted g^ir in section 2.14 of RFC 7296. It is always exactly 32 octets when Curve25519 is used.

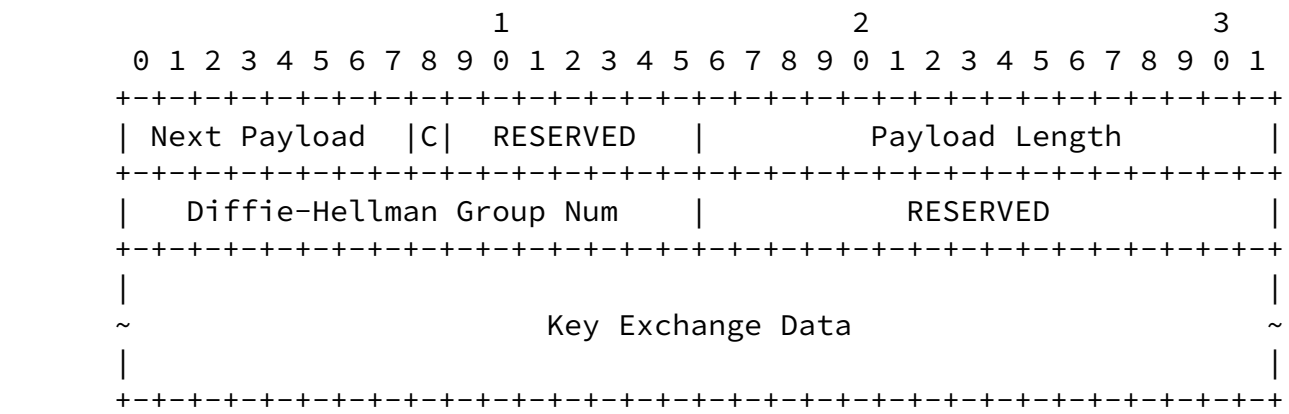
A complete description of the Curve25519 function, as well as a few implementation notes, are provided in Appendix A.

3. Use and Negotiation in IKEv2

The use of Curve25519 in IKEv2 is negotiated using a Transform Type 4 (Diffie-Hellman group) in the SA payload of either an IKE_SA_INIT or a CREATE_CHILD_SA exchange.

3.1. Key Exchange Payload

The diagram for the Key Exchange Payload from section 3.4 of RFC 7296 is copied below for convenience:



- Payload Length - Since a Curve25519 public key is 32 octets, the Payload Length is always 40.
- The Diffie-Hellman Group Num is xx for Curve25519 (TBA by IANA)
- The Key Exchange Data is 32 octets encoded as an array of bytes in little-endian order as described in section 8 of [CFRG-Curves]

3.2. Recipient Tests

This section describes the checks that a recipient of a public key needs to perform. It is the equivalent of the tests described in [RFC6989] for other Diffie-Hellman groups.

Curve25519 was designed in a way that the result of Curve25519(x, d) will never reveal information about d, provided is was chosen as prescribed, for any value of x.

Define legitimate values of x as the values that can be obtained as x

= Curve25519(G, d') for some d, and call the other values illegitimate. The definition of the Curve25519 function shows that legitimate values all share the following property: the high-order bit of the last byte is not set.

Since there are some implementation of the Curve25519 function that impose this restriction on their input and others that don't, implementations of Curve25519 in IKE SHOULD reject public keys when the high-order bit of the last byte is set (in other words, when the value of the leftmost byte is greater than 0x7F) in order to prevent implementation fingerprinting.

Other than this recommended check, implementations do not need to ensure that the public keys they receive are legitimate: this is not necessary for security with Curve25519.

4. Security Considerations

Curve25519 is designed to facilitate the production of high-performance constant-time implementations of the Curve25519 function. Implementors are encouraged to use a constant-time implementation of the Curve25519 function. This point is of crucial importance if the implementation chooses to reuse its supposedly ephemeral key pair for many key exchanges, which some implementations do in order to improve performance.

Curve25519 is believed to be at least as secure as the 256-bit random ECP group (group 19) defined in RFC 4753, also known as NIST P-256. While the NIST curves are advertised as being chosen verifiably at random, there is no explanation for the seeds used to generate them. In contrast, the process used to pick Curve25519 is fully documented and rigid enough so that independent verification has been done. This is widely seen as a security advantage for Curve25519, since it prevents the generating party from maliciously manipulating the parameters.

Another family of curves available in IKE, generated in a fully verifiable way, is the Brainpool curves [RFC6954]. Specifically,

brainpoolP256 (group 28) is expected to provide a level of security comparable to Curve25519 and NIST P-256. However, due to the use of pseudo-random prime, it is significantly slower than NIST P-256, which is itself slower than Curve25519.

5. IANA Considerations

IANA is requested to assign one value from the IKEv2 "Transform Type 4 - Diffie-Hellman Group Transform IDs" registry, with name Curve25519, and this document as reference. The Recipient Tests field should also point to this document.

6. Acknowledgements

Curve25519 was designed by D. J. Bernstein and Tanja Lange. The specification of wire format is by Sean Turner, Rich Salz, and Watson Ladd, with Adam Langley editing the current document. Much of the text in this document is copied from Simon's draft for the TLS working group.

7. References

7.1. Normative References

[CFRG-Curves] Langley, A., "Elliptic Curves for Security", draft-agl-cfrgcurve-00 (work in progress), January 2015.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC7296] Kivinen, T., Kaufman, C., Hoffman, P., Nir, Y., and P. Eronen, "Internet Key Exchange Protocol Version 2 (IKEv2)", RFC 7296, October 2014.

7.2. Informative References

[Curve25519]
Bernstein, J., "Curve25519: New Diffie-Hellman Speed Records", LNCS 3958, February 2006, <http://dx.doi.org/10.1007/11745853_14>.

[EFD] Bernstein, D. and T. Lange, "Explicit-Formulas Database: XZ coordinates for Montgomery curves", January 2014, <http://www.hyperelliptic.org/EFD/g1p/auto-montgom-xz.html>.

[NaCl] Bernstein, D., "Cryptography in NaCL", March 2013, <http://cr.yp.to/highspeed/naclcrypto-20090310.pdf>.

[RFC4753] Fu, D. and J. Solinas, "ECP Groups For IKE and IKEv2", RFC 4753, January 2007.

[RFC6954] Merkle, J. and M. Lochter, "Using the Elliptic Curve Cryptography (ECC) Brainpool Curves for the Internet Key Exchange Protocol Version 2 (IKEv2)", RFC 6954, July 2013.

[RFC6989] Sheffer, Y. and S. Fluhrer, "Additional Diffie-Hellman Tests for the Internet Key Exchange Protocol Version 2 (IKEv2)", RFC 6989, July 2013.

Appendix A. The curve25519 function

A.1. Formulas

This section completes Section 2 by defining the Curve25519 function and the common public key G. It is meant as an alternative, self-contained specification for the Curve25519 function, possibly easier to follow than the original paper for most implementors.

A.1.1. Field Arithmetic

Throughout this section, P denotes the integer 2^255-19 = 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFED. The letters X and Z, and their numbered variants such as x1, z2, etc. denote integers modulo P, that is integers between 0 and P-1 and every operation between them is implictly done modulo P. For addition, subtraction and multiplication this means doing the

operation in the usual way and then replacing the result with the remainder of its division by P. For division, "X / Z" means mutliplying (mod P) X by the modular inverse of Z mod P.

A convenient way to define the modular inverse of Z mod P is as Z^(P-2) mod P, that is Z to the power of 2^255-21 mod P. It is also a practical way of computing it, using a square-and-multiply method.

The four operations +, -, *, / modulo P are known as the field operations. Techniques for efficient implementation of the field operations are outside the scope of this document.

A.1.2. Conversion to and from internal format

For the purpose of this section, we will define a Curve25519 point as a pair (X, Z) were X and Z are integers mod P (as defined above). Though public keys were defined to be strings of 32 bytes, internally they are represented as curve points. This subsection describes the conversion process as two functions: PubkeyToPoint and PointToPubkey.

```
PubkeyToPoint:
Input: a public key b_0, ..., b_31
Output: a Curve25519 point (X, Z)
    1. Set X = b_0 + 256 * b_1 + ... + 256^31 * b_31 mod P
    2. Set Z = 1
    3. Output (X, Z)

PointToPubkey:
Input: a Curve25519 point (X, Z)
Output: a public key b_0, ..., b_31
    1. Set x1 = X / Z mod P
    2. Set b_0, ... b_31 such that
        x1 = b_0 + 256 * b_1 + ... + 256^31 * b_31 mod P
    3. Output b_0, ..., b_31
```

A.1.3. Scalar Multiplication

We first introduce the DoubleAndAdd function, defined as follows (formulas taken from [EFD]).


```
DoubleAndAdd:
```

```
Input: two points (X2, Z2), (X3, Z3), and an integer mod P: X1
Output: two points (X4, Z4), (X5, Z5)
Constant: the integer mod P: a24 = 121666 = 0x01DB42
Variables: A, AA, B, BB, E, C, D, DA, CB are integers mod P
    1. Do the following computations mod P:
        A  = X2 + Z2
        AA = A2
        B  = X2 - Z2
        BB = B2
        E  = AA - BB
        C  = X3 + Z3
        D  = X3 - Z3
        DA = D * A
        CB = C * B
        X5 = (DA + CB)^2
        Z5 = X1 * (DA - CB)^2
        X4 = AA * BB
        Z4 = E * (BB + a24 * E)
    2. Output (X4, Z4) and (X5, Z5)
```

This may be taken as the abstract definition of an arbitrary-looking function. However, let's mention "the true meaning" of this function, without justification, in order to help the reader make more sense of it. It is possible to define operations "+" and "-" between Curve25519 points. Then, assuming (X2, Z2) - (X3, Z3) = (X1, 1), the DoubleAndAdd function returns points such that (X4, Z4) = (X2, Z2) + (X2, Z2) and (X5, Z5) = (X2, Z2) + (X3, Z3).

Taking the "+" operation as granted, we can define multiplication of a Curve25519 point by a positive integer as N * (X, Z) = (X, Z) + ... + (X, Z), with N point additions. It is possible to compute this operation, known as scalar multiplication, using an algorithm called the Montgomery ladder, as follows.

```
ScalarMult:
Input: a Curve25519 point: (X, 1) and a 255-bits integer: N
Output: a point (X1, Z1)
Variable: a point (X2, Z2)
    0. View N as a sequence of bits b_254, ..., b_0,
        with b_254 the most significant bit
        and b_0 the least significant bit.
    1. Set X1 = 1 and Z1 = 0
    2. Set X2 = X and Z2 = 1
    3. For i from 254 downwards to 0, do:
        If b_i == 0, then:
            Set (X2, Z2) and (X1, Z1) to the output of
            DoubleAndAdd((X2, Z2), (X1, Z1), X)
        else:
            Set (X1, Z1) and (X2, Z2) to the output of
            DoubleAndAdd((X1, Z1), (X2, Z2), X)
    4. Output (X1, Z1)
```

A.1.4. Conclusion

We are now ready to define the Curve25519 function itself.

```
Curve25519:
Input: a public key P and a secret key S
Output: a public key Q
Variables: two Curve25519 points (X, Z) and (X1, Z1)
    1. Set (X, Z) = PubkeyToPoint(P)
    2. Set (X1, Z1) = ScalarMult((X, Z), S)
    3. Set Q = PointToPubkey((X1, Z1))
    4. Output Q
```

The common public key G mentioned in the first paragraph of Section 2 is defined as G = PointToPubkey((9, 1).

A.2. Test vectors

The following test vectors are taken from [NaCl]. Compared to this reference, the private key strings have been applied the ClampC function of the reference and converted to integers in order to fit the description given in [Curve25519] and the present memo.

The secret key of party A is denoted by S_a, it public key by P_a, and similarly for party B. The shared secret is SS.

```
S_a = 0x6A2CB91DA5FB77B12A99C0EB872F4CDF
        4566B25172C1163C7DA518730A6D0770

P_a = 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

S_b = 0x6BE088FF278B2F1CFDB6182629B13B6F
        E60E80838B7FE1794B8A4A627E08AB58

P_b = 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

 SS = 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
```

A.3. Side-channel considerations

Curve25519 was specifically designed so that correct, fast, constant-time implementations are easier to produce. In particular, using a Montgomery ladder as described in the previous section ensures that, for any valid value of the secret key, the same sequence of field operations are performed, which eliminates a major source of side-channel leakage.

However, merely using Curve25519 with a Montgomery ladder does not prevent all side-channels by itself, and some point are the responsibility of implementors:

1. In step 3 of SclarMult, avoid branches depending on b_i, as well as memory access patterns depending on b_i, for example by using safe conditional swaps on the inputs and outputs of DoubleAndAdd.
2. Avoid data-dependant branches and memory access patterns in the implementation of field operations.

Techniques for implementing the field operations in constant time and with high performance are out of scope of this document. Let's mention however that, provided constant-time multiplication is available, division can be computed in constant time using exponentiation as described in Appendix A.1.1.

If using constant-time implementations of the field operations is not convenient, an option to reduce the information leaked this way is to replace step 2 of the SclarMult function with:

2a. Pick Z uniformly randomly between 1 and P-1 included
2b. Set X2 = X * Z and Z2 = Z

This method is known as randomizing projective coordinates. However, it is no guaranteed to avoid all side-channel leaks related to field operations.

Side-channel attacks are an active reseach domain that still sees new significant results, so implementors of the Curve25519 function are advised to follow recent security research closely.

Authors' Addresses

Yoav Nir
Check Point Software Technologies Ltd.
5 Hasolelim st.
Tel Aviv 6789735
Israel

Email: ynir.ietf@gmail.com

Simon Josefsson
SJD AB

Email: simon@josefsson.org