Using Algebraic Eraser (AEDH) in OpenPGP
draft-atkins-openpgp-algebraic-eraser-05

Abstract

The Algebraic Eraser(TM) is an encryption engine that supports, among other configurations, a Diffie-Hellman-like key agreement protocol. This draft specifies how to encode, store, share, and use Algebraic Eraser Key Agreement Protocol (AEKAP, also called AEDH) keys in OpenPGP.

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 20, 2015.

Copyright Notice

Copyright (c) 2015 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 (http://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.
1. Introduction

The OpenPGP specification in [RFC4880] defines the use of RSA, Elgamal, and DSA public key algorithms. [RFC6637] adds support for Elliptic Curve Cryptography and specifies the ECDSA and ECDH algorithms.

The Algebraic Eraser(TM) was first introduced in Key agreement, the Algebraic Eraser, and lightweight cryptography [AAGL] published by the American Mathematical Society in 2004. It describes "a new key agreement protocol suitable for implementation on low-cost platforms which constrain the use of computational resources." It is further compared to other algorithms in [AEIntro]. This document specifies how to encode, store, and use the Algebraic Eraser Key Agreement Protocol (AEKAP, also called AEDH) in OpenPGP.

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. The Algebraic Eraser
The Algebraic Eraser brings together the Braid Group, Matrices, and operations over small Finite Fields to produce an algorithm that executes linear in time with the increase in key size.

A complete description of the Algebraic Eraser is available in [AAGL].

2.1. E-Multiplication

The Algebraic Eraser defines an operation called "E-Multiplication" upon which the algorithm is based (see [AAGL]). E-Multiplication (denoted herein by *) takes one matrix (M0) and permutation (S0) and operates on a second matrix (M1) and permutation (S1), resulting in another matrix (M2) and permutation (S2). In other words: (M0,S0) * (M1,S1) = (M2,S2).

The secret to E-multiplication is that you can take a Braid element (an "Artin generator") and map that to a matrix and permutation upon which you can operate. Specifically, each Artin generator is associated to a specific Colored Burau (CB) matrix and permutation (see [AAGL]). The E-multiplication process involves permuting the variables in the CB matrix using the current permutation, substituting those variables with the TValues from the Keyset (see Section 2.2), then multiplying that against the starting matrix resulting in the another matrix. Then you multiply the permutations, resulting in a new permutation. This new matrix and permutation is the result of the E-Multiplication.

For the purposes of clarity, the permutation is recorded as numbers from 0 to N-1. Braid generators are recorded from 0 (for b1) to N-2 (for b(n-1)) with an extra bit used to signify inverse. The matrix field elements (and tvalues) are recorded from 0 to q-1.

2.2. AEDH Keyset Parameters

AEDH Keyset Parameters are similar to Diffie-Hellman cyclic groups of prime order or ECC curves. Just as users must choose the same DH prime or ECC curve in order to communicate, similarly participants in the AEDH must be using the same Keyset Parameters.

The first basic set of parameters is the chosen Braid Group and Field Size, BnFq, where n is the number of strands in the chosen braid (also called the braid index) and q is the size of the field in use. The field size, q, must be a power of a prime. Generally it is 2^r (where r is a small integer) although this is not a requirement. For example, one might choose B10F8 or B16F32. This is like choosing how many bits to use when generating a prime for Diffie-Hellman.
Once the BnFq space is chosen then the Keyset Parameters can be
generated by a trusted third party (TTP). First they generate an
n-by-n matrix (M) where each entry in the matrix is a member of the
field Fq. Second, the TTP generates a set of TValues, which is an
array of n invertable entries within the field Fq (i.e., values 1 to
Fq-1). Finally, the TTP generates at least two sets of braid
conjugates, Ca and Cb, where each conjugate in Ca commutes with each
conjugate in Cb. The conjugates are lists of "braid words", or
"Artin generators" within the Bn braid group. The TTP generates La
conjugates for set Ca and Lb conjugates for set Cb, where the numbers
La and Lb MAY be different.

The public Keyset Parameters are the Matrix (M), TValues array, and
conjugate sets and must be available to generate keys that can
communicate. Moreover, the TValue array must be available to anyone
using the Keyset. These Keysets MAY be published and named, but MUST
be numbered with an OID.

For two users to execute the AEDH they MUST generate keys from the
same Keyset and they MUST choose from different conjugate sets within
that Keyset. I.e., for Alice and Bob to complete the AEDH Alice must
generate her key from Ca and Bob must generate his key from Cb.

This document does not specify any particular Keyset Parameters that
MUST be implemented.

2.3. Generating Key Pairs

The Algebraic Eraser has a two-part Private Key and a two-part Public
Key. The Public key is then generated from the two Private Keys.

To generate the 1st private key you generate a random polynomial and
apply that to the public matrix from the keyset within the keyset
field. This results in an n-by-n matrix where each entry in the
matrix is a member of the field Fq (although the last row of the
matrix contains n-1 zeros). The key search space for the 1st private
key is q^(n-1). Note that the 1st private key permutation is always
the identity permutation, so there is no need to store it.

To generate the 2nd private key you choose a random set of conjugates
(and inverses) and string them together. This results in a long
string of Artin generators (and inverses). You MAY reduce the string
if you so choose using the Dehornoy reduction [Dehornoy]. The search
space of the 2nd private key is (2l)^k (where l is the number of
published conjugates (La or Lb), and k the number of chosen
conjugates and inverses).
The Public Key is computed by an E-Multiplication of the 1st private key and the 2nd private key, where the 2nd private key is iteratively processed (see Section 2.1. The result (the public key) is an n-by-n matrix of Fq and another permutation.

Note that the last row of the Public Key Matrix is all zero except for the last entry. When encoding the Public Key you SHOULD ignore those zeros.

2.4. Computing the Shared Secret

To compute the shared secret you first perform regular matrix multiplication of the 1st private key against the matrix component of the public key you receive from the other person. This results in another matrix. The permutation of the 1st private key is the identity, hence the result of multiplying it against the permutation of the public key leaves the permutation of the public key unchanged and you can just use it directly. Then you perform E-multiplication of this matrix/permutation result against the 2nd private key. The resulting matrix/permutation is the shared secret.

3. Encoding of Public and Private Keys

Each portion of a key can be reduced to a byte-string (or, more accurately, multiple byte strings). Each matrix can be encoded by stringing together each field element in each row and then stringing each row together. A permutation can be encoded by stringing together each element in the list. The conjugates are also encoded by stringing together each element.

The following public key algorithm IDs are added to expand section 9.1 of [RFC4880], "Public-Key Algorithms":

<table>
<thead>
<tr>
<th>ID</th>
<th>Description of Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBD1</td>
<td>AEDH public key algorithm</td>
</tr>
</tbody>
</table>

Encoding of Public and Private keys MUST use the version 4 packet format (or newer).
3.1. Encoding Bit-Strings

The Algebraic eraser uses matrices, fields, and braids that are denoted in bits, particular strings of bits. These objects need to be encoded into bit strings for storage and transmission. While the most simplistic encoding method is to take each field as a byte (or multi-byte word) and string them together, that would waste a lot of space due to the sparse nature of the entries. Instead the objects should be encoded in a bit-packed method.

3.1.1. Bit Packing

The most economical method to encode Algebraic Eraser elements is "bit packing", which implies dropping all unused bits and merging adjacent fields into a single bit string. Obviously if the bit width is a direct power of two then packing into 8-bit bytes is simple. However if the field bit width is an odd length then bit-packing provides to most economical method.

Bit packing merges all bits together and then cuts it up into 8-bit bytes. For example if the entries are 5 bits each you use 3 bits from the second entry to merge into the first, then shift the remaining 2 bits of the second entry, combine with the next 5 bits from the third, and then one bit from the fourth entry, and so on, until you’ve reached the end. This method could end up with unused bits at the end of the string which must be zero.

Assume you require five (5) bits to encode your numbers, the following table shows how you could could use bit packing to encode four entries (where a, b, c, and d are the bits in the first, second, third, and fourth entries):

<table>
<thead>
<tr>
<th>Full Bytes:</th>
<th>000aaaaa</th>
<th>000bbbbbb</th>
<th>000cccccc</th>
<th>000dddddd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit packing:</td>
<td>aaaaabbb</td>
<td>bbcccccd</td>
<td>dddd0000</td>
<td></td>
</tr>
</tbody>
</table>

Any unused bits MUST be left as zero (and MUST be checked to be zero).

Recall that matrices in AEDH have a row of zeros; this row SHOULD be assumed to "not exist". When using bit packing you SHOULD just tack the last entry of the final row onto the end of the list of entries of the rest of the matrix. This could result in an odd number of entries depending on your n and q choices which could result is extra (zero) bits at the end of the packed string.
3.1.2. Multi-Byte Entries

In the case of entries wider than 8 bits (e.g. a Field parameter greater than 256), the bits are combined in network byte order. However they can still be merged together using bit packing from Section 3.1.1 (in the case of entries that are not 8-bit multiples). For example, a 12-bit field (F4096) could be combined a nibble at a time, or a 10-bit field (F1024) could use bit-packing.

3.2. Encoding Public Keys

The following algorithm specific packets are added to Section 5.5.2 of [RFC4880], "Public-Key Packet Formats", to support AEDH:

- a variable length field containing a keyset parameter OID, formatted as follows (see [RFC6637] for a full description of the OID encoding method):
  - a one-octet size of the following field; values 0 and 0xFF are reserved for future extensions,
  - octets representing a keyset parameter OID
- one byte denoting from which set of conjugates in the keyset this key was generated (e.g. the Alice set or the Bob set)
- MPI of the public key matrix
- MPI of the public key permutation

3.3. Encoding Private Keys

The following algorithm specific packets are added to Section 5.5.3 of [RFC4880], "Secret-Key Packet Formats", to support AEDH:

- MPI of the 1st private key (matrix)
- MPI of the 2nd private key (conjugate string)

4. Acknowledgements

The term "Algebraic Eraser" is a trademark of SecureRF Corporation and is used herein with permission.
The author would like to thank Paul Gunnells, Dorian Goldfeld, and Iris Anshel for their tireless efforts to review this document, suggest improvements, and explain to me how to improve my description of how AE works. Big thanks also to Werner Koch and Vedaal for their comments and suggestions.

5. IANA Considerations

IANA is requested to assign an algorithm number from the OpenPGP Public-Key Algorithms range, or the "namespace" in the terminology of [RFC5226], that was created by [RFC4880]. See Section 3.

<table>
<thead>
<tr>
<th>ID</th>
<th>Algorithm</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBD1</td>
<td>AEDH public key algorithm</td>
<td>This doc</td>
</tr>
</tbody>
</table>

[Notes to RFC-Editor: Please remove the table above on publication. It is desirable not to reuse old or reserved algorithms because some existing tools might print a wrong description. A higher number is also an indication for a newer algorithm. As of now 22 is the next free number, but we request the selection of 23.]

6. Security Considerations

The security considerations of [RFC4880] apply accordingly.

AEDH will generate the same session key when used with the same two public/private key pairs. The authors of AE generally recommend that at least one party use an ephemeral key pair in order to prevent the same session key being generated every time.

AEDH is an encryption-only algorithm, therefore it cannot self-certify a key. To have an AEDH master key you MUST implement [I-D.atkins-openpgp-device-certificates].

When using the generated session key, you MUST only use the bits included in the protocol. You should MUST NOT use any always-zero bits, including those in the last row of the matrix.

7. References

7.1. Normative References

[AAGL]  Anshel, I., Anshel, M., Goldfeld, D., and S. Lemieux, "Key agreement, the Algebraic Eraser, and lightweight
www.securerf.com/wp-content/uploads/2014/03/SecureRF-

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate


[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an
IANA Considerations Section in RFCs", BCP 26, RFC 5226,
May 2008.

[RFC6637] Jivsov, A., "Elliptic Curve Cryptography (ECC) in
OpenPGP", RFC 6637, June 2012.

7.2. Informative References

[AEIntro] SecureRF Corporation, SRF., "An Introduction to
Cryptographic Security Methods and Their Role in Securing

Advances in Mathematics 123, 1997,

draft-atkins-openpgp-device-certificates-00 (work in
progress), August 2014.

Appendix A. Test Vectors

To help implementing this specification a non-normative example is
provided. This example assumes:

- the algorithm id for AEDH will be 23
- the keyset OID 1.3.6.1.4.1.44196.1.0.2, which defines:
  * the braid/field as B10F256
  * the TValues are (decimal): 29 102 76 50 33 235 255 244 64 143
and gets encoded with length 11 and the following hex bytes: 2B 06 01 04 01 82 D9 24 01 00 02

A.1. Sample key

The secret key used for this example is:

1st Private Key Matrix:

36 202 16 154 154 39 148 101 59 49
100 4 153 41 114 92 96 189 183 205
76 154 186 93 85 142 19 154 93 232
221 202 113 59 48 68 31 217 43 14
62 241 82 90 133 185 137 46 131 3
112 66 234 91 153 13 244 127 23 254
219 141 237 2 9 113 56 97 82 203
57 223 142 245 143 176 30 139 53 160
220 199 106 211 27 207 77 195 114 76
0 0 0 0 0 0 0 0 0 0 90

2nd Private key (bit-packed, first two bytes encode the generator count):

02 39 3c e9 59 4c 85 31 e6 49 0e 85 b5 f1 42 98 e8 a0 22 08 c8 64
ad 8e 42 0e 53 a1 4d 29 c6 43 21 4d 29 90 b6 25 67 29 4e 91 10 c7
18 c4 51 90 02 10 88 40 08 00 00 04 01 14 c0 19 4c 01 94 82 08 c4
40 0c 86 10 88 60 08 82 19 04 22 18 00 10 88 42 18 44 11 0c 01 10
80 00 04 21 10 80 10 84 42 08 86 00 80 22 10 c0 11 0c 43 08 44 11
4c 32 1c a6 11 04 43 10 c2 08 ca 32 04 44 00 84 42 10 84 29 84 32
00 23 20 02 20 08 84 21 0a 60 0c 80 19 4e 61 0c 87 29 86 43 18 64
31 80 22 10 a7 30 8a 62 04 63 29 82 32 90 e3 18 c6 31 10 e3 21 82
22 9c 64 08 ca 32 00 42 29 80 32 1c a6 00 c8 60 88 a7 10 06 52 9c
e7 30 46 52 88 87 19 4a 40 88 a7 39 82 31 8c 62 08 8a 71 04 62 00
80 32 1c 81 11 4e 32 98 23 19 0e 32 94 c1 19 4e 60 8c 44 38 ca 60
00 64 30 44 43 8c 81 10 44 53 04 64 08 88 71 90 23 20 46 22 8c 64
00 88 11 94 c0 19 4a 52 18 c0 08 c4 42 98 23 21 80 21 08 42 10 c8
63 00 23 29 4e 32 9c a6 00 80 22 18 02 10 84 21 08 80 08 ca 52 94
a5 20 04 40 0c 86 31 92 53 a5 6d 70 84 42 1d 2b 4a d8 f8 1d 0a 6b
a0 43 00 42 29 90 b6 3c c8 00 c8 64 a9 ae 4a da e4 90 45 3a 54 00
08 87 42 8c 82 18

The key was created on 2015-06-18 16:21:39 EDT from the tag conjugates (type 1), and thus the fingerprint of the OpenPGP key is:

E587 BC12 2480 EF5B 6142 7C8F 94B1 5FDA 09D3 B5D4

and the entire public key packet is:
A.2. Sample key agreement

The key agreement is created using the sample key against a second (reader) public key. The reader public key has the following data:

Matrix (bit-packed):

   2d 8b 8f 63 a2 61 7b 58  51 3e a6 c9 6c af 41 41
   a3 18 ce 05 d6 95 54 40  12 95 a1 c5 3f 79 b8 29
   94 a6 36 af e4 79 14 c3  95 dd 78 c7 7e a0 57 7d
   1b 92 d0 94 46 48 65 a0  a1 f0 5f 65 6b a5 db b5
  61 32 df 5d f9 ae 79 16 c1 38 f0 50 a0 4a 81 f4
  33 04 71 8b b7 ba 1b 73  36 87 f3

Permutation (packed): 01 35 24 67 89

Which results in the following shared secret:

Matrix:

   70 251 120  22 113  68 157 233  68 221
  142 124  37 172 199 212  32 202 188 110
  184 18  66 215 212  32 178 138 161 184
  179 139  46 238  57 103  69  6  73 240
  81 255 231 127 142 152 139  10 146  98
  188  7 152  13 207 247  0 192 148  35
  226 210  97 101 162  39 188 179 248 198
  103 36 169 112 201  60  95 122 250 210
  173 18 246 202 155 145  4 102 189 192
   0 0 0 0 0 0 0 0 0 31

Permutation (decimal): 8 1 3 5 2 4 0 7 9 6

Author’s Address
Derek Atkins
SecureRF Corporation
100 Beard Sawmill Rd, Suite 350
Shelton, CT  06484
US

Phone: +1 617 623 3745
Email: datkins@securerf.com, derek@ihtfp.com