Internet Research Task Force (IRTF)
Request for Comments: 7664
Category: Informational
ISSN: 2070-1721

D. Harkins, Ed.
Aruba Networks

November 2015

Dragonfly Key Exchange

Abstract

This document specifies a key exchange using discrete logarithm cryptography that is authenticated using a password or passphrase. It is resistant to active attack, passive attack, and offline dictionary attack. This document is a product of the Crypto Forum Research Group (CFRG).

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Research Task Force (IRTF). The IRTF publishes the results of Internet-related research and development activities. These results might not be suitable for deployment. This RFC represents the individual opinion(s) of one or more members of the Crypto Forum Research Group of the Internet Research Task Force (IRTF). Documents approved for publication by the IRSG are not a candidate for any level of Internet Standard; see Section 2 of RFC 5741.

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/rfc7664.

Table of Contents

1. Introduction

Passwords and passphrases are the predominant way of doing authentication in the Internet today. Many protocols that use passwords and passphrases for authentication exchange password-derived data as a proof-of-knowledge of the password (for example, [RFC7296] and [RFC5433]). This opens the exchange up to an offline dictionary attack where the attacker gleans enough knowledge from either an active or passive attack on the protocol to run through a pool of potential passwords and compute verifiers until it is able to match the password-derived data.

This protocol employs discrete logarithm cryptography to perform an efficient exchange in a way that performs mutual authentication using a password that is provably resistant to an offline dictionary attack. Consensus of the CFRG for this document was rough.

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].

1.2. Definitions

1.2.1. Notations

The following notations are used in this memo.

password
: A shared, secret, and potentially low-entropy word, phrase, code, or key used as a credential to mutually authenticate the peers. It is not restricted to characters in a human language.

a | b
: denotes concatenation of bit string "a" with bit string "b".

len(a)
: indicates the length in bits of the bit string "a".

lsb(a)
: returns the least-significant bit of the bit string "a".

lgr(a,b)
: takes "a" and a prime, "b", and returns the Legendre symbol (a/b).

min(a,b)
: returns the lexicographical minimum of strings "a" and "b", or zero (0) if "a" equals "b".

max(a,b)
: returns the lexicographical maximum of strings "a" and "b", or zero (0) if "a" equals "b".

The convention for this memo is to represent an element in a finite cyclic group with an uppercase letter or acronym, while a scalar is indicated with a lowercase letter or acronym. An element that represents a point on an elliptic curve has an implied composite nature -- i.e., it has both an x- and y-coordinate.

1.2.2. Resistance to Dictionary Attack

Resistance to dictionary attack means that any advantage an adversary can gain must be directly related to the number of interactions she makes with an honest protocol participant and not through computation. The adversary will not be able to obtain any information about the password except whether a single guess from a protocol run is correct or incorrect.

2. Discrete Logarithm Cryptography

Dragonfly uses discrete logarithm cryptography to achieve authentication and key agreement (see [SP800-56A]). Each party to the exchange derives ephemeral keys with respect to a particular set of domain parameters (referred to here as a "group"). A group can be based on Finite Field Cryptography (FFC) or Elliptic Curve Cryptography (ECC).

Three operations are defined for both types of groups:

- "scalar operation" -- takes a scalar and an element in the group to produce another element -- Z = scalar-op(x, Y).
- "element operation" -- takes two elements in the group to produce a third -- Z = element-op(X, Y).
- "inverse operation" -- takes an element and returns another element such that the element operation on the two produces the identity element of the group -- Y = inverse(X).

2.1. Elliptic Curve Cryptography

Domain parameters for the ECC groups used by Dragonfly are:

- A prime, p, determining a prime field GF(p). The cryptographic group will be a subgroup of the full elliptic curve group that consists of points on an elliptic curve -- elements from GF(p) that satisfy the curve's equation -- together with the "point at infinity" that serves as the identity element. The group operation for ECC groups is addition of points on the elliptic curve.
- Elements a and b from GF(p) that define the curve's equation. The point (x, y) in GF(p) x GF(p) is on the elliptic curve if and only if (y^2 - x^3 - a*x - b) mod p equals zero (0).
- A point, G, on the elliptic curve, which serves as a generator for the ECC group. G is chosen such that its order, with respect to elliptic curve addition, is a sufficiently large prime.
- A prime, q, which is the order of G, and thus is also the size of the cryptographic subgroup that is generated by G.

An (x,y) pair is a valid ECC element if: 1) the x- and y-coordinates are both greater than zero (0) and less than the prime defining the underlying field; and, 2) the x- and y-coordinates satisfy the equation for the curve and produce a valid point on the curve that is

not the point at infinity. If either one of those conditions do not hold, the (x,y) pair is not a valid element.

The scalar operation is addition of a point on the curve with itself a number of times. The point Y is multiplied x times to produce another point Z:

```
Z = scalar-op(x, Y) = x*Y
```

The element operation is addition of two points on the curve. Points X and Y are summed to produce another point Z:

```
Z = element-op(X, Y) = X + Y
```

The inverse function is defined such that the sum of an element and its inverse is "0", the point at infinity of an elliptic curve group:

```
R + inverse(R) = "0"
```

Elliptic curve groups require a mapping function, q = F(Q), to convert a group element to an integer. The mapping function used in this memo returns the x-coordinate of the point it is passed.

scalar-op(x, Y) can be viewed as x iterations of element-op() by defining:

```
Y = scalar-op(1, Y)

Y = scalar-op(x, Y) = element-op(Y, scalar-op(x-1, Y)), for x > 1
```

A definition of how to add two points on an elliptic curve (i.e., element-op(X, Y)) can be found in [RFC6090].

Note: There is another elliptic curve domain parameter, a cofactor, h, that is defined by the requirement that the size of the full elliptic curve group (including "0") be the product of h and q. Elliptic curve groups used with Dragonfly authentication MUST have a cofactor of one (1).

2.2. Finite Field Cryptography

Domain parameters for the FFC groups used in Dragonfly are:

- A prime, p, determining a prime field GF(p), the integers modulo p. The FFC group will be a subgroup of GF(p)*, the multiplicative group of non-zero elements in GF(p). The group operation for FFC groups is multiplication modulo p.

- An element, G, in GF(p)* which serves as a generator for the FFC group. G is chosen such that its multiplicative order is a sufficiently large prime divisor of ((p-1)/2).

- A prime, q, which is the multiplicative order of G, and thus also the size of the cryptographic subgroup of GF(p)* that is generated by G.

A number is a valid element in an FFC group if: 1) it is between one (1) and one (1) less than the prime, p, exclusive (i.e., 1 < element < p-1); and, 2) if modular exponentiation of the element by the group order, q, equals one (1). If either one of those conditions do not hold, the number is not a valid element.

The scalar operation is exponentiation of a generator modulo a prime. An element Y is taken to the x-th power modulo the prime returning another element, Z:

```
Z = scalar-op(x, Y) = Y^x mod p
```

The element operation is modular multiplication. Two elements, X and Y, are multiplied modulo the prime returning another element, Z:

```
Z = element-op(X, Y) = (X * Y) mod p
```

The inverse function for a MODP group is defined such that the product of an element and its inverse modulo the group prime equals one (1). In other words,

```
(R * inverse(R)) mod p = 1
```

3. The Dragonfly Key Exchange

There are two parties to the Dragonfly exchange named, for convenience and by convention, Alice and Bob. The two parties have a shared password that was established in an out-of-band mechanism, and they both agree to use a particular domain parameter set (either ECC or FFC). In the Dragonfly exchange, both Alice and Bob share an identical view of the shared password -- i.e., it is not "augmented", where one side holds a password and the other side holds a non-invertible verifier. This allows Dragonfly to be used in traditional client-server protocols and also in peer-to-peer applications in which there are not fixed roles and either party may initiate the exchange (and both parties may implement it simultaneously).

Prior to beginning the Dragonfly exchange, the two peers MUST derive a secret element in the chosen domain parameter set. Two "hunting-and-pecking" techniques to determine a secret element, one for ECC

and one for FFC, are described in Section 3.2, but any secure, deterministic method that is agreed upon can be used. For instance, the technique described in [hash2ec] can be used for ECC groups.

The Dragonfly exchange consists of two message exchanges, a "Commit Exchange" in which both sides commit to a single guess of the password, and a "Confirm Exchange" in which both sides confirm knowledge of the password. A side effect of running the Dragonfly exchange is an authenticated, shared, and secret key whose cryptographic strength is set by the agreed-upon group.

Dragonfly uses a random function, H(), a mapping function, F(), and a key derivation function, KDF().

3.1. Assumptions

In order to avoid attacks on the Dragonfly protocol, some basic assumptions are made:

1. Function H is a "random oracle" (see [RANDOR]) that maps a binary string of indeterminate length onto a fixed binary string that is x bits in length.

   H: {0,1}^* --> {0,1}^x

2. Function F is a mapping function that takes an element in a group and returns an integer. For ECC groups, function F() returns the x-coordinate of the element (which is a point on the elliptic curve); for FFC groups, function F() is the identity function (since all elements in an FFC group are already integers less than the prime).

   ECC: x = F(P), where P=(x,y)

   FFC: x = F(x)

3. Function KDF is a key derivation function (see, for instance, [SP800-108]) that takes a key to stretch, k, a label to bind to the key, label, and an indication of the desired output, n:

   stretch = KDF-n(k, label)

   so that len(stretch) equals n.

4. The discrete logarithm problem for the chosen group is hard. That is, given G, P, and Y = G^x mod p, it is computationally infeasible to determine x. Similarly, for an ECC group given the curve definition, a generator G, and Y = x * G, it is computationally infeasible to determine x.

5. There exists a pool of passwords from which the password shared by the two peers is drawn. This pool can consist of words from a dictionary, for example. Each password in this pool has an equal probability of being the shared password. All potential attackers have access to this pool of passwords.

6. The peers have the ability to produce quality random numbers.

3.2. Derivation of the Password Element

Prior to beginning the exchange of information, the peers MUST derive a secret element, called the Password Element (PE), in the group defined by the chosen domain parameter set. From the point of view of an attacker who does not know the password, the PE will be a random element in the negotiated group. Two examples are described here for completeness, but any method of deterministically mapping a secret string into an element in a selected group can be used -- for instance, the technique in [hash2ec] for ECC groups. If a different technique than the ones described here is used, the secret string SHOULD include the identities of the peers.

To fix the PE, both peers MUST have a common view of the password. If there is any password processing necessary (for example, to support internationalization), the processed password is then used as the shared credential. If either side wants to store a hashed version of the password (hashing the password with random data called a "salt"), it will be necessary to convey the salt to the other side prior to commencing the exchange, and the hashed password is then used as the shared credential.

Note: Only one party would be able to maintain a salted password, and this would require that the Dragonfly key exchange be used in a protocol that has strict roles for client (that always initiates) and server (that always responds). Due to the symmetric nature of Dragonfly, salting passwords does not prevent an impersonation attack after compromise of a database of salted passwords.

The deterministic process to select the PE begins with choosing a secret seed and then performing a group-specific hunting-and-pecking technique -- one for FFC groups and another for ECC groups.

To thwart side-channel attacks that attempt to determine the number of iterations of the hunting-and-pecking loop used to find the PE for a given password, a security parameter, k, is used that ensures that at least k iterations are always performed. The probability that one requires more than n iterations of the hunting-and-pecking loop to find an ECC PE is roughly (q/2p)^n and to find an FFC PE is roughly (q/p)^n, both of which rapidly approach zero (0) as n increases. The security parameter, k, SHOULD be set sufficiently large such that the probability that finding the PE would take more than k iterations is sufficiently small (see Section 4).

First, an 8-bit counter is set to one (1), and a secret base is computed using the negotiated one-way function with the identities of the two participants, Alice and Bob, the secret password, and the counter:

base = H(max(Alice,Bob) | min(Alice,Bob) | password | counter)

The identities are passed to the max() and min() functions to provide the necessary ordering of the inputs to H() while still allowing for a peer-to-peer exchange where both Alice and Bob each view themselves as the "initiator" of the exchange.

The base is then stretched using the technique from Section B.5.1 of [FIPS186-4]. The key derivation function, KDF, is used to produce a bitstream whose length is equal to the length of the prime from the group’s domain parameter set plus the constant sixty-four (64) to derive a temporary value, and the temporary value is modularly reduced to produce a seed:

n = len(p) + 64

temp = KDF-n(base, "Dragonfly Hunting and Pecking")

seed = (temp mod (p - 1)) + 1

The string bound to the derived temporary value is for illustrative purposes only. Implementations of the Dragonfly key exchange SHOULD use a usage-specific label with the KDF.

Note: The base is stretched to 64 more bits than are needed so that the bias from the modular reduction is not so apparent.

The seed is then passed to the group-specific hunting-and-pecking technique.

If the protocol performing the Dragonfly exchange has the ability to exchange random nonces, those SHOULD be added to the computation of the base to ensure that each run of the protocol produces a different PE.

3.2.1. Hunting and Pecking with ECC Groups

The ECC-specific hunting-and-pecking technique entails looping until a valid point on the elliptic curve has been found. The seed is used as an x-coordinate with the equation of the curve to check whether x^3 + a*x + b is a quadratic residue modulo p. If it is not, then the counter is incremented, a new base and new seed are generated, and the hunting and pecking continues. If it is a quadratic residue modulo p, then the x-coordinate is assigned the value of seed and the current base is stored. When the hunting-and-pecking loop terminates, the x-coordinate is used with the equation of the curve to solve for a y-coordinate. An ambiguity exists since two values for the y-coordinate would be valid, and the low-order bit of the stored base is used to unambiguously determine the correct y-coordinate. The resulting (x,y) pair becomes the Password Element, PE.

Algorithmically, the process looks like this:

```
found = 0
counter = 1
n = len(p) + 64
do {
  base = H(max(Alice,Bob) | min(Alice,Bob) | password | counter)
  temp = KDF-n(base, "Dragonfly Hunting And Pecking")
  seed = (temp mod (p - 1)) + 1
  if ( (seed^3 + a*seed + b) is a quadratic residue mod p)
  then
    if ( found == 0 )
    then
      x = seed
      save = base
      found = 1
    fi
  fi
  counter = counter + 1
} while ((found == 0) || (counter <= k))
y = sqrt(x^3 + ax + b)
if ( lsb(y) == lsb(save) )
then
  PE = (x,y)
else
  PE = (x,p-y)
fi
```

Figure 1: Fixing PE for ECC Groups

Checking whether a value is a quadratic residue modulo a prime can leak information about that value in a side-channel attack. Therefore, it is RECOMMENDED that the technique used to determine if the value is a quadratic residue modulo p blind the value with a random number so that the blinded value can take on all numbers between 1 and p-1 with equal probability while not changing its quadratic residuosity. Determining the quadratic residue in a fashion that resists leakage of information is handled by flipping a coin and multiplying the blinded value by either a random quadratic residue or a random quadratic nonresidue and checking whether the multiplied value is a quadratic residue (qr) or a quadratic nonresidue (qnr) modulo p, respectively. The random residue and nonresidue can be calculated prior to hunting and pecking by calculating the Legendre symbol on random values until they are found:

```
do {
  qr = random() mod p
} while ( lgr(qr, p) != 1)

do {
  qnr = random() mod p
} while ( lgr(qnr, p) != -1)
```

Algorithmically, the masking technique to find out whether or not a value is a quadratic residue looks like this:

```
is_quadratic_residue (val, p) {
    r = (random() mod (p - 1)) + 1
    num = (val * r * r) mod p
    if ( lsb(r) == 1 )
       num = (num * qr) mod p
       if ( lgr(num, p) == 1)
       then
          return TRUE
       fi
    else
       num = (num * qnr) mod p
       if ( lgr(num, p) == -1)
       then
          return TRUE
       fi
    fi
    return FALSE
}
```

3.2.2. Hunting and Pecking with MODP Groups

The MODP-specific hunting-and-pecking technique entails finding a random element which, when used as a generator, will create a group with the same order as the group created by the generator from the domain parameter set. The secret generator is found by exponentiating the seed to the value ((p-1)/q), where p is the prime and q is the order from the domain parameter set. If that value is greater than one (1), it becomes the PE; otherwise, the counter is incremented, a new base and seed are generated, and the hunting and pecking continues.

Algorithmically, the process looks like this:

```
found = 0
counter = 1
n = len(p) + 64
do {
  base = H(max(Alice,Bob) | min(Alice,Bob) | password | counter)
  temp = KDF-n(seed, "Dragonfly Hunting And Pecking")
  seed = (temp mod (p - 1)) + 1
  temp = seed ^ ((p-1)/q) mod p
  if (temp > 1)
  then
    if (not found)
      PE = temp
      found = 1
    fi
  fi
  counter = counter + 1
} while ((found == 0) || (counter <= k))
```

Figure 2: Fixing PE for MODP Groups

3.3. The Commit Exchange

In the Commit Exchange, both sides commit to a single guess of the password. The peers generate a scalar and an element, exchange them with each other, and process the other's scalar and element to generate a common and shared secret.

First, each peer generates two random numbers, private and mask that are each greater than one (1) and less than the order from the selected domain parameter set:

```
1 < private < q
1 < mask < q
```

These two secrets and the Password Element are then used to construct the scalar and element:

```
scalar = (private + mask) modulo q
Element = inverse(scalar-op(mask, PE))
```

If the scalar is less than two (2), the private and mask MUST be thrown away and new values generated. Once a valid scalar and Element are generated, the mask is no longer needed and MUST be irretrievably destroyed.

The peers exchange their scalar and Element and check the peer's scalar and Element, deemed peer-scalar and Peer-Element. If the peer has sent an identical scalar and Element -- i.e., if scalar equals peer-scalar and Element equals Peer-Element -- it is sign of a reflection attack, and the exchange MUST be aborted. If the values differ, peer-scalar and Peer-Element must be validated. For the peer-scalar to be valid, it MUST be between 1 and q exclusive. Validation of the Peer-Element depends on the type of cryptosystem -- validation of an (x,y) pair as an ECC element is specified in Section 2.1, and validation of a number as an FFC element is specified in Section 2.2. If either the peer-scalar or Peer-Element fail validation, then the exchange MUST be terminated and authentication fails. If both the peer-scalar and Peer-Element are valid, they are used with the Password Element to derive a shared secret, ss:

```
ss = F(scalar-op(private,
                 element-op(peer-Element,
                            scalar-op(peer-scalar, PE))))
```

To enforce key separation and cryptographic hygiene, the shared secret is stretched into two subkeys -- a key confirmation key, kck, and a master key, mk. Each of the subkeys SHOULD be at least the length of the prime used in the selected group.

```
kck | mk = KDF-n(ss, "Dragonfly Key Derivation")
```

where n = len(p)*2.

3.4. The Confirm Exchange

In the Confirm Exchange, both sides confirm that they derived the same secret, and therefore, are in possession of the same password.

The Commit Exchange consists of an exchange of data that is the output of the random function, H(), the key confirmation key, and the two scalars and two elements exchanged in the Commit Exchange. The order of the scalars and elements are: scalars before elements, and sender's value before recipient's value. So from each peer's perspective, it would generate:

```
confirm = H(kck | scalar | peer-scalar |
            Element | Peer-Element | <sender-id>)
```

Where <sender-id> is the identity of the sender of the confirm message. This identity SHALL be that contributed by the sender of the confirm message in generation of the base in Section 3.2.

The two peers exchange these confirmations and verify the correctness of the other peer's confirmation that they receive. If the other peer's confirmation is valid, authentication succeeds; if the other peer's confirmation is not valid, authentication fails.

If authentication fails, all ephemeral state created as part of the particular run of the Dragonfly exchange MUST be irretrievably destroyed. If authentication does not fail, mk can be exported as an authenticated and secret key that can be used by another protocol, for instance IPsec, to protect other data.

4. Security Considerations

The Dragonfly exchange requires both participants to have an identical representation of the password. Salting of the password merely generates a new credential -- the salted password -- that must be identically represented on both sides. If an adversary is able to gain access to the database of salted passwords, she would be able to impersonate one side to the other, even if she was unable to determine the underlying, unsalted password.

Resistance to dictionary attack means that an adversary must launch an active attack to make a single guess at the password. If the size of the dictionary from which the password was extracted was d, and each password in the dictionary has an equal probability of being chosen, then the probability of success after a single guess is 1/d. After x guesses, and removal of failed guesses from the pool of possible passwords, the probability becomes 1/(d-x). As x grows, so does the probability of success. Therefore, it is possible for an adversary to determine the password through repeated brute-force, active, guessing attacks. Users of the Dragonfly key exchange SHOULD ensure that the size of the pool from which the password was drawn, d, is sufficiently large to make this attack preventable. Implementations of Dragonfly SHOULD support countermeasures to deal with this attack -- for instance, by refusing authentication attempts for a certain amount of time, after the number of failed authentication attempts reaches a certain threshold. No such threshold or amount of time is recommended in this memo.

Due to the problems with using groups that contain a small subgroup, it is RECOMMENDED that implementations of Dragonfly not allow for the specification of a group's complete domain parameter to be sent in-line, but instead use a common repository and pass an identifier to a domain parameter set whose strength has been rigorously proven and that does not have small subgroups. If a group's complete domain parameter set is passed in-line, it SHOULD NOT be used with Dragonfly unless it directly matches a known good group.

It is RECOMMENDED that an implementation set the security parameter, k, to a value of at least forty (40) which will put the probability that more than forty iterations are needed in the order of one in one trillion (1:1,000,000,000,000).

The technique used to obtain the Password Element in Section 3.2.1 addresses side-channel attacks in a manner deemed controversial by some reviewers in the CFRG. An alternate method, such as the one defined in [hash2ec], can be used to alleviate concerns.

This key exchange protocol has received cryptanalysis in [clarkehao]. [lanskro] provides a security proof of Dragonfly in the random oracle model when both identities are included in the data sent in the Confirm Exchange (see Section 3.4).

5. References

5.1. 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>.

5.2. Informative References

[clarkehao] Clarke, D. and F. Hao, "Cryptanalysis of the Dragonfly Key Exchange Protocol", IET Information Security, Volume 8, Issue 6, DOI 10.1049/iet-ifs.2013.0081, November 2014.

[FIPS186-4] NIST, "Digital Signature Standard (DSS)", Federal Information Processing Standard (FIPS) 186-4, DOI 10.6028/NIST.FIPS.186-4, July 2013.

[hash2ec] Brier, E., Coron, J-S., Icart, T., Madore, D., Randriam, H., and M. Tibouchi, "Efficient Indifferentiable Hashing into Ordinary Elliptic Curves", Cryptology ePrint Archive Report 2009/340, 2009.

[lanskro] Lancrenon, J. and M. Skrobot, "On the Provable Security of the Dragonfly Protocol", Proceedings of 18th International Information Security Conference (ISC 2015), pp 244-261, DOI 10.1007/978-3-319-23318-5_14, September 2015.

[RANDOR] Bellare, M. and P. Rogaway, "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols", Proceedings of the 1st ACM Conference on Computer and Communication Security, ACM Press, DOI 10.1145/168588.168596, 1993.

[RFC5433] Clancy, T. and H. Tschofenig, "Extensible Authentication Protocol - Generalized Pre-Shared Key (EAP-GPSK) Method", RFC 5433, DOI 10.17487/RFC5433, February 2009, <http://www.rfc-editor.org/info/rfc5433>.

[RFC6090] McGrew, D., Igoe, K., and M. Salter, "Fundamental Elliptic Curve Cryptography Algorithms", RFC 6090, DOI 10.17487/RFC6090, February 2011, <http://www.rfc-editor.org/info/rfc6090>.

[RFC7296] Kaufman, C., Hoffman, P., Nir, Y., Eronen, P., and T. Kivinen, "Internet Key Exchange Protocol Version 2 (IKEv2)", STD 79, RFC 7296, DOI 10.17487/RFC7296, October 2014, <http://www.rfc-editor.org/info/rfc7296>.

[SP800-108] Chen, L., "Recommendation for Key Derivation Using Pseudorandom Functions", NIST Special Publication 800-108, October 2009.

[SP800-56A] Barker, E., Johnson, D., and M. Smid, "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography (Revised)", NIST Special Publication 800-56A, March 2007.

Acknowledgements

The author would like to thank Kevin Igoe and David McGrew, chairmen of the Crypto Forum Research Group (CFRG) for agreeing to accept this memo as a CFRG work item. Additional thanks go to Scott Fluhrer and Hideyuki Suzuki for discovering attacks against earlier versions of this key exchange and suggesting fixes to address them. Lily Chen provided helpful discussions on hashing into an elliptic curve. Rich Davis suggested the validation steps used on received elements to prevent a small subgroup attack. Dylan Clarke and Feng Hao discovered a dictionary attack against Dragonfly if those checks are not made and a group with a small subgroup is used. And finally, a very heartfelt thanks to Jean Lancrenon and Marjan Skrobot for developing a proof of the security of Dragonfly.

The blinding scheme to prevent side-channel attacks when determining whether a value is a quadratic residue modulo a prime was suggested by Scott Fluhrer. Kevin Igoe suggested addition of the security parameter k to hide the amount of time taken hunting and pecking for the password element.

Author’s Address

Dan Harkins (editor)
Aruba Networks
1322 Crossman Avenue
Sunnyvale, CA 94089-1113
United States

Email: dharkins@arubanetworks.com