Internet Engineering Task Force   D. Harkins, Ed.
Internet-Draft   Aruba Networks
Intended status: Standards Track   D. Halasz, Ed.
Expires: September 9, 2012   Halasz Ventures
                                                          March 8, 2012

          Secure Password Ciphersuites for Transport Layer Security (TLS)
                          draft-harkins-tls-pwd-02

Status of this Memo

Copyright Notice

Abstract

   This memo defines several new ciphersuites for the Transport Layer
   Security (TLS) protocol to support certificate-less, secure

   authentication using only a simple, low-entropy, password. The
   ciphersuites are all based on an authentication and key exchange
   protocol that is resistant to off-line dictionary attack.

Table of Contents

1. Background

1.1. The Case for Certificate-less Authentication

   TLS usually uses public key certificates for authentication
   [RFC5246]. This is problematic in some cases:

   o Frequently, TLS [RFC5246] is used in devices owned, operated, and
      provisioned by people who lack competency to properly use
      certificates and merely want to establish a secure connection
      using a more natural credential like a simple password. The
      proliferation of deployments that use a self-signed server
      certificate in TLS [RFC5246] followed by a PAP-style exchange over
      the unauthenticated channel underscores this case.

   o A password is a more natural credential than a certificate (from
      early childhood people learn the semantics of a shared secret), so
      a password-based TLS ciphersuite can be used to protect an HTTP-
      based certificate enrollment scheme-- e.g. an [RFC5967] -style
      request and an [RFC5751] -style response-- to parlay a simple
      password into a certificate for subsequent use with any
      certificate-based authentication protocol. This addresses a
      significant "chicken-and-egg" dilemma found with certificate-only
      use of [RFC5246].

   o Some PIN-code readers will transfer the entered PIN to a smart
      card in clear text. Assuming a hostile environment, this is a bad
      practice. A password-based TLS ciphersuite can enable the
      establishment of an authenticated connection between reader and
      card based on the PIN.

1.2. Resistance to Dictionary Attack

   It is a common misconception that a protocol that authenticates with
   a shared and secret credential is resistent to dictionary attack if
   the credential is assumed to be an N-bit uniformly random secret,
   where N is sufficiently large. The concept of resistence to
   dictionary attack really has nothing to do with whether that secret
   can be found in a standard collection of a language's defined words
   (i.e. a dictionary). It has to do with how an adversary gains an
   advantage in attacking the protocol.

   For a protocol to be resistant to dictionary attack any advantage an
   adversary can gain must be a function of the amount of interactions
   she makes with an honest protocol participant and not a function of
   the amount of computation she uses. The adversary will not be able
   to obtain any information about the password except whether a single
   guess from a single protocol run which she took part in is correct or

incorrect.

It is assumed that the attacker has access to a pool of data from which the secret was drawn-- it could be all numbers between 1 and 2^N, it could be all defined words in a dictionary. The key is that the attacker cannot do a an attack and then enumerate through the pool trying potential secrets (computation) to see if one is correct. She must do an active attack for each secret she wishes to try (interaction) and the only information she can glean from that attack is whether the secret used with that particular attack is correct or not.

2. Keyword Definitions

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].

3. Introduction

3.1. Notation

The following notation is used in this memo:

password
    a secret, and potentially low-entropy word, phrase, code or key used as a credential for authentication. The password is shared between the TLS client and TLS server.

$y = H(x)$
    a binary string of arbitrary length, x, is given to a function H which produces a fixed-length output, y.

a | b
    denotes concatenation of string a with string b.

[a]b
    indicates a string consisting of the single bit "a" repeated "b" times.

x mod y
    indicates the remainder of division of x by y. The result will be between 0 and y.

   LSB(x)
        returns the least-significant bit of the bitstring "x".

3.2. Discrete Logarithm Cryptography

   The ciphersuites defined in this memo use discrete logarithm
   cryptography (see [SP800-56A]) to produce an authenticated and shared
   secret value that is an element in a group defined by a set of domain
   parameters. The domain parameters can be based on either Finite
   Field Cryptography (FFC) or Elliptic Curve Cryptography (EEC).

   Elements in a group, either an FFC or EEC group, are indicated using
   upper-case while scalar values are indicated using lower-case.

3.2.1. Elliptic Curve Cryptography

   The authenticated key exchange defined in this memo uses fundamental
   algorithms of elliptic curves defined over GF(p) as described in
   [RFC6090].

   Domain parameters for the ECC groups used by this memo are:

   o A prime, p, determining a prime field GF(p). The cryptographic
      group will be a subgroup of the full elliptic curve group which
      consists 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.

   o 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).

   o 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.

   o A prime, q, which is the order of G, and thus is also the size of
      the cryptographic subgroup that is generated by G.

   o A co-factor, f, defined by the requirement that the size of the
      full elliptic curve group (including the "point at infinity") is
      the product of f and q.

   This memo uses the following ECC Functions:

   o Z = elem-op(X,Y) = X + Y: two points on the curve X and Y, are
      sumed to produce another point on the curve, Z. This is the group
      operation for ECC groups.

o Z = scalar-op(x,Y) = x * Y: an integer scalar, x, acts on a point
   on the curve, Y, via repetitive addition (Y is added to itself x
   times), to produce another EEC element, Z.

o Y = inverse(X): a point on the curve, X, has an inverse, Y, which
   is also a point on the curve, when their sum is the "point at
   infinity" (the identity for elliptic curve addition). In other
   words, R + inverse(R) = "0".

o z = F(X): the x-coordinate of a point (x, y) on the curve is
   returned. This is a mapping function to convert a group element
   into an integer.

Only ECC groups over GF(p) can be used with TLS-PWD. ECC groups over
GF(2^m) SHALL NOT be used by TLS-PWD. In addition, ECC groups with a
co-factor greater than one (1) SHALL NOT be used by TLS-PWD.

A composite (x, y) pair can be validated as an a point on the
elliptic curve by checking whether: 1) both coordinates x and y are
greater than zero (0) and less than the prime defining the underlying
field; 2) the x- and y- coordinates satisfy the equation of the
curve; and 3) they do not represent the point-at-infinity "0". If
any of those conditions are not true the (x, y) pair is not a valid
point on the curve.

3.2.2. Finite Field Cryptography

Domain parameters for the FFC groups used by this memo are:

o 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).

o 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).

o 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.

This memo uses the following FFC Functions:

o Z = elem-op(X,Y) = (X * Y) mod p: two FFC elements, X and Y, are
   multiplied modulo the prime, p, to produce another FFC element, Z.
   This is the group operation for FFC groups.

   o Z = scalar-op(x,Y) = Y^x mod p: an integer scalar, x, acts on an
      FFC group element, Y, via exponentiation modulo the prime, p, to
      produce another FFC element, Z.

   o Y = inverse(X): a group element, X, has an inverse, Y, when the
      product of the element and its inverse modulo the prime equals one
      (1). In other words, (X * inverse(X)) mod p = 1.

   o z = F(X): is the identity function since an element in an FFC
      group is already an integer. It is included here for consistency
      in the specification.

   Many FFC groups used in IETF protocols are based on safe primes and
   do not define an order (q). For these groups, the order (q) used in
   this memo shall be the prime of the group minus one divided by two--
   (p-1)/2.

   An integer can be validated as being an element in an FFC group by
   checking whether: 1) it is between one (1) and the prime, p,
   exclusive; and 2) if modular exponentiation of the integer by the
   group order, q, equals one (1). If either of these conditions are
   not true the integer is not an element in the group.

3.3. Instantiating the Random Function

   The protocol described in this memo uses a random function, H, which
   is modeled as a "random oracle". At first glance, one may view this
   as a hash function. As noted in [RANDOR], though, hash functions are
   too structured to be used directly as a random oracle. But they can
   be used to instantiate the random oracle.

   The random function, H, in this memo is instantiated by using the
   hash algorithm defined by the particular TLS-PWD ciphersuite in HMAC
   mode with a key whose length is equal to block size of the hash
   algorithm and whose value is zero. For example, if the ciphersuite
   is TLS_ECCPWD_WITH_AES_128_GCM_SHA256 then H will be instantiated
   with SHA256 as:

       H(x) = HMAC-SHA256([0]32, x)

3.4. Passwords

   The authenticated key exchange used in TLS-PWD requires each side to
   have a common view of a shared credential. To protect a database of
   stored passwords, though, the password SHALL be salted and the
   result, called the base, SHALL be used as the authentication
   credential.

The salting function is defined as:

    base = HMAC-SHA256(salt, username | password)

The password used for generation of the base SHALL be represented as
a UTF-8 encoded character string processed according to the rules of
the [RFC4013] profile of [RFC3454] and the salt SHALL be a 32 octet
random number. The server SHALL store a triplet of the form:

    { username, base, salt }

And the client SHALL generate the base upon receiving the salt from
the server.

3.5. Assumptions

The security properties of the authenticated key exchange defined in
this memo are based on a number of assumptions:

1. The random function, H, is a "random oracle" as defined in
    [RANDOR].

2. The discrete logarithm problem for the chosen group is hard.
    That is, given g, p, and $y = g^x \bmod p$, 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.

3. Quality random numbers with sufficient entropy can be created.
    This may entail the use of specialized hardware. If such
    hardware is unavailable a cryptographic mixing function (like a
    strong hash function) to distill enropy from multiple,
    uncorrelated sources of information and events may be needed. A
    very good discussion of this can be found in [RFC4086].

4. Specification of the TLS-PWD Handshake

The authenticated key exchange is accomplished by each side deriving
a password-based element, PE, in the chosen group, making a
"committment" to a single guess of the password using PE, and
generating the Premaster Secret. The ability of each side to produce
a valid finished message authenticates itself to the other side.

The authenticated key exchange is dropped into the standard TLS
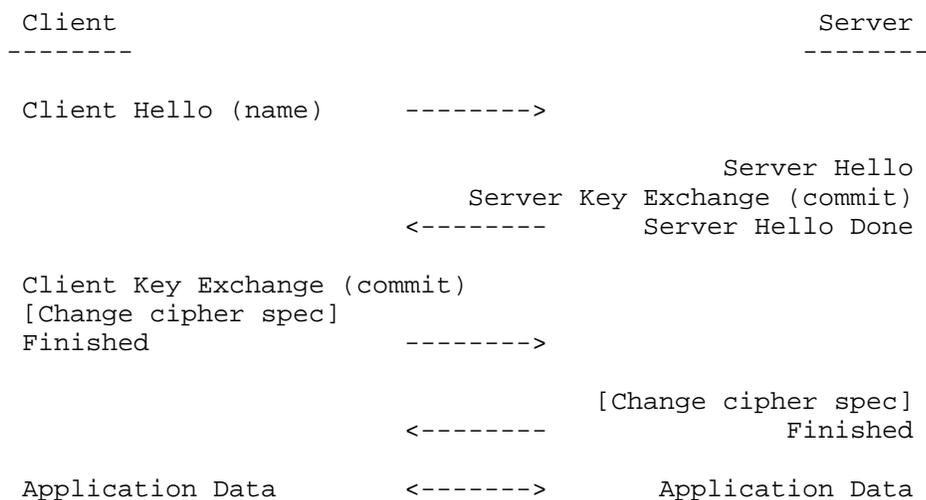message handshake by modifying some of the messages.

```
        Client                                            Server
        --------                                          --------

        Client Hello (name)       -------->

                                                      Server Hello
                                  Server Key Exchange (commit)
                                  <--------        Server Hello Done

        Client Key Exchange (commit)
        [Change cipher spec]
        Finished                  -------->

                                                  [Change cipher spec]
                                  <--------                  Finished

        Application Data          <------->         Application Data
```

                               Figure 1

4.1. Fixing the Password Element

   Prior to making a "committment" both sides must generate a secret
   element, PE, in the chosen group using the common password-derived
   base. The server generates PE after it receives the Client Hello and
   chooses the particular group to use, and the client generates PE upon
   receipt of the Server Key Exchange.

   Fixing the password element involves an iterative "hunting and
   pecking" technique using the prime from the negotiated group's domain
   parameter set and an ECC- or FFC-specific operation depending on the
   negotiated group.

   To thwart side channel attacks which attempt to determine the number
   of iterations of the "hunting-and-pecking" loop are used to find PE
   for a given password, a security parameter, k, is used to ensure that
   at least k iterations are always performed. This technique need only
   be used with ECC groups.

   First, an 8-bit counter is set to the value one (1). Then, H is used
   to generate a password seed from the a counter, the prime of the
   selected group, and the base (which is derived from the username,
   password, and salt):

                   pwd-seed = H(base | counter | p)

   Then, the pwd-seed is expanded using the PRF to the length of the
   prime from the negotiated group's domain parameter set, to create

   pwd-value:

      pwd-value = PRF(pwd-seed, "TLS-PWD Hunting And Pecking",
                      ClientHello.random | ServerHello.random) [0..p];

   If the pwd-value is greater than or equal to the prime, p, the
   counter is incremented, and a new pwd-seed is generated and the
   hunting-and-pecking continues. If pwd-value is less than the prime,
   p, it is passed to the group-specific operation which either returns
   the selected password element or fails. If the group-specific
   operation fails, the counter is incremented, a new pwd-seed is
   generated, and the hunting-and-pecking continues. This process
   continues until the group-specific operation returns the password
   element. For FCC groups, this terminates the hunting-and-pecking
   process. For ECC groups, after the password element has been chosen,
   the base is changed to a random number, the counter is incremented
   and the hunting-and-pecking continues until the counter is greater
   than the security parameter, k.

   When PE has been discovered, pwd-seed and pwd-value SHALL be
   irretrievably destroyed.

4.1.1. Computing an ECC Password Element

   The group-specific operation for ECC groups uses pwd-value, pwd-seed,
   and the equation for the curve to produce PE. First, pwd-value is
   used directly as the x-coordinate, x, with the equation for the
   elliptic curve, with parameters a and b from the domain parameter set
   of the curve, to solve for a y-coordinate, y. If there is no
   solution to the quadratic equation, this operation fails and the
   hunting-and-pecking process continues. If a solution is found, then
   an ambiguity exists as there are technically two solutions to the
   equation and pwd-seed is used to unambiguously select one of them.
   If the low-order bit of pwd-seed is equal to the low-order bit of y,
   then a candidate PE is defined as the point (x, y); if the low-order
   bit of pwd-seed differs from the low-order bit of y, then a candidate
   PE is defined as the point (x, p - y), where p is the prime over
   which the curve is defined. The candidate PE becomes PE, a random
   number is used instead of the base, and the hunting and pecking
   continues until it has looped through k iterations.

   Algorithmically, the process looks like this:

```
      found = 0
      counter = 0
      base = H(username | password | salt)
      do {
         counter = counter + 1
         pwd-seed = H(base | counter | p)
         pwd-value = PRF(pwd-seed, "TLS-PWD Hunting And Pecking",
                         ClientHello.random | ServerHello.random) [0..p]
         if (pwd-value < p)
         then
            x = pwd-value
            if ( (y = sqrt(x^3 + ax + b)) != FAIL)
              then
              if (found == 0)
              then
                 if (LSB(y) == LSB(pwd-seed))
                 then
                    PE = (x, y)
                 else
                    PE = (x, p-y)
                 fi
                 found = 1
              else
                 base = random()
              fi
           fi
         fi
      } while ((found == 0) || (counter <= k))
```

                   Figure 2: Fixing PE for ECC Groups

   The probability that one requires more than "n" iterations of the
   "hunting and pecking" loop to find PE is roughly (q/2p)^n which
   rapidly approaches zero (0) as "n" increases. Therefore the security
   parameter, k, SHOULD be set sufficiently large such that the
   probability that finding PE would take more than k iterations is
   sufficiently small.

4.1.2. Computing an FFC Password Element

   The group-specific operation for FFC groups takes pwd-value, and the
   prime, p, and order, q, from the group's domain parameter set (see
   Section 3.2.2 when the order is not part of the defined domain
   parameter set) to directly produce a candidate password element, by
   exponentiating the pwd-value to the value ((p-1)/q) modulo the prime.
   If the result is greater than one (1), the candidate password element
   becomes PE, and the hunting and pecking terminates successfully.

Algorithmically, the process looks like this:

```
found = 0
counter = 0
do {
  counter = counter + 1
  pwd-seed = H(base | counter | p)
  pwd-value = PRF(pwd-seed, "TLS-PWD Hunting And Pecking",
                  ClientHello.random | ServerHello.random) [0..p]
  if (pwd-value < p)
  then
    PE = pwd-value ^ ((p-1)/q) mod p
    if (PE > 1)
    then
      found = 1
    fi
  fi
} while (found == 0)
```

Figure 3: Fixing PE for FFC Groups

4.2. Changes to Handshake Message Contents

4.2.1. Client Hello Changes

The client is required to identify herself to the server by adding a
PWD extension to the Client Hello message. The PWD extension uses
the standard mechanism defined in [RFC5246]. The "extension data"
field of the PWD extension SHALL contain a PWD_name which is used to
identify the password shared between the client and server.

```
enum { pwd(TBD) } ExtensionType;

opaque PWD_name<1..2^8-1>;
```

The PWD_name SHALL be UTF-8 encoded character string processed
according to the rules of the [RFC4013] profile of [RFC3454].

A client offering a PWD ciphersuite MUST include the PWD extension in
her Client Hello.

If a server does not have a password identified by the PWD_name in
the PWD extension of the Client Hello, the server SHOULD hide that
fact by simulating the protocol-- putting random data in the PWD-
specific components of the Server Key Exchange-- and then rejecting
the client's finished message with a "bad_record_mac" alert. To
properly effect a simulated TLS-PWD exchange, an appropriate delay
SHOULD be inserted between receipt of the Client Hello and response

of the Server Hello. Alternately, a server MAY choose to terminate
the exchange if a password identified by the PWD_name in the PWD
extension of the Client Hello is not found.

The server decides on a group to use with the named user (see
Section 9 and generates the password element, PE, according to
Section 4.1.2.

4.2.2. Server Key Exchange Changes

The domain parameter set for the selected group MUST be specified in
the ServerKeyExchange, either explicitly or, in the case of some
elliptic curve groups, by name. In addition to the group
specification, the ServerKeyExchange also contains the server's
"committment" in the form of a scalar and element, and the salt which
was used to store the user's password.

Two new values have been added to the enumerated KeyExchangeAlgorithm
to indicate TLS-PWD using finite field cryptography, ff_pwd, and TLS-
PWD using elliptic curve cryptography, ec_pwd.

```
            enum { ff_pwd, ec_pwd } KeyExchangeAlgorithms;

            struct {
              opaque salt<1..2^8-1>;
              opaque pwd_p<1..2^16-1>;
              opaque pwd_g<1..2^16-1>;
              opaque pwd_q<1..2^16-1>;
              opaque ff_sscalar<1..2^16-1>;
              opaque ff_selement<1..2^16-1>;
            } ServerFFPWDParams;

            struct
              opaque salt<1..2^8-1>;
              ECParameters curve_params;
              opaque ec_sscalar<1..2^8-1>;
              ECPoint ec_selement;
            } ServerECPWDParams;

            struct {
              select (KeyExchangeAlgorithm) {
                case ec_pwd:
                  ServerECPWDParams params;
                case ff_pwd:
                  ServerFFPWDParams params;
              };
            } ServerKeyExchange;
```

4.2.2.1. Generation of ServerKeyExchange

   The scalar and Element that comprise the server's "committment" are
   generated as follows.

   First two random numbers, called private and mask, between zero and
   the order of the group (exclusive) are generated. If their sum
   modulo the order of the group, q, equals zero the numbers must be
   thrown away and new random numbers generated. If their sum modulo
   the order of the group, q, is greater than zero the sum becomes the
   scalar.

        scalar = (private + mask) mod q

   The Element is then calculated as the inverse of the group's scalar
   operation (see the group specific operations in Section 3.2) with the
   mask and PE.

        Element = inverse(scalar-op(mask, PE))

   After calculation of the scalar and Element the mask SHALL be
   irretrievably destroyed.

4.2.2.1.1. ECC Server Key Exchange

   EEC domain parameters are specified, either explicitly or named, in
   the ECParameters component of the EEC-specific ServerKeyExchange as
   defined in [RFC4492]. The scalar SHALL become the ec_sscalar
   component and the Element SHALL become the ec_selement of the
   ServerKeyExchange. If the client requested a specific point format
   (compressed or uncompressed) with the Support Point Formats Extension
   (see [RFC4492]) in its Client Hello, the Element MUST be formatted in
   the ec_selement to conform to that request.

   As mentioned in Section 3.2.1, elliptic curves over GF(2^m), so
   called characteristic-2 curves, and curves with a co-factor greater
   than one (1) SHALL NOT be used with TLS-PWD.

4.2.2.1.2. FFC Server Key Exchange

   FFC domain parameters sent in the ServerKeyExchange are for the
   group's prime, generator (which is only used for verification of the
   group specification), and the order of the group's generator. The
   scalar SHALL become the ff_sscalar component and the Element SHALL
   become the ff_selement in the FFC-specific ServerKeyExchange.

   As mentioned in Section 3.2.2 if the prime is a safe prime and no
   order is included in the domain parameter set, the order added to the

ServerKeyExchange SHALL be the prime minus one divided by two--
(p-1)/2.

4.2.2.2. Processing of ServerKeyExchange

Upon receipt of the ServerKeyExchange, the client decides whether to
support the indicated group or not. Named elliptic curves are easy
to validate-- either they are supported or they are not, but care
must be taken with FFC groups and explicitly specified ECC groups.
As mentioned in Section 3.5, the discrete logarithm problem MUST be
hard for any group used with this memo. The specific steps taken to
come to this assurance for a particular group are outside the scope
of this memo but they are the same steps to take when using the
Diffie-Hellman key exchange with TLS. If the client decides not to
support the group indicated in the ServerKeyExchange, she MUST abort
the exchange.

If the client decides to support the indicated group the server's
"commitment" MUST be validated by ensuring that: 1) the server's
scalar value is greater than zero (0) and less than the order of the
group, q; and 2) that the Element is valid for the chosen group (see
Section 3.2.2 and Section 3.2.1 for how to determine whether an
Element is valid for the particular group. Note that if the Element
is a compressed point on an elliptic curve it MUST be uncompressed
before checking its validity).

If the group is acceptable, the client extracts the salt from the
ServerKeyExchange and generates the password element, PE, according
to Section 3.4 and Section 4.1.2.

4.2.3. Client Key Exchange Changes

When the value of KeyExchangeAlgorithm is either ff_pwd or ec_pwd,
the ClientKeyExchange is used to convey the client's "committment" to
the server. It, therefore, contains a scalar and an Element.

```
                     struct {
                       opaque ff_cscalar<1..2^16-1>;
                       opaque ff_celement<1..2^16-1>;
                     } ClientFFPWDParams;

                     struct
                       opaque ec_cscalar<1..2^8-1>;
                       ECPoint ec_celement;
                     } ClientECPWDParams;

                     struct {
                       select (KeyExchangeAlgorithm) {
                         case ff_pwd: ClientFFPWDParams;
                         case ec_pwd: ClientECPWDParams;
                       } exchange_keys;
                     } ClientKeyExchange;
```

4.2.3.1. Generation of Client Key Exchange

   The client's scalar and Element are generated in the manner described
   in Section 4.2.2.1.

   For an FFC group, the scalar SHALL become the ff_cscalar component
   and the Element SHALL become the ff_celement in the FFC-specific
   ClientKeyExchange.

   For an ECC group, the scalar SHALL become the ec_cscalar component
   and the ELement SHALL become the ec_celement in the ECC-specific
   ClientKeyExchange. If the client requested a specific point format
   (compressed or uncompressed) with the Support Point Formats Extension
   in its ClientHello, then the Element MUST be formatted in the
   ec_celement to conform to its initial request.

4.2.3.2. Processing of Client Key Exchange

   The server MUST validate the client's "committment" by ensuring that:
   1) the client's scalar value is greater than zero (0) and less than
   the order of the group, q; and 2) that the Element is valid for the
   chosen group (see Section 3.2.2 and Section 3.2.1 for how to determin
   whether an Element is valid for a particular group. Note that if the
   Element is a compressed point on an elliptic curve it MUST be
   uncompressed before checking its validity.

4.3. Computing the Premaster Secret

   The client uses her own scalar and Element, denoted here
   ClientKeyExchange.scalar and ClientKeyExchange.Element, the server's
   scalar and Element, denoted here as ServerKeyExchange.scalar and

ServerKeyExchange.Element, and the random private value, denoted here
as client.private, she created as part of the generation of her
"commit" to compute an intermediate value, z, as indicated:

```
z = F(scalar-op(client.private,
             element-op(ServerKeyExchange.Element,
                         scalar-op(ServerKeyExchange.scalar, PE))))
```

With the same notation as above, the server uses his own scalar and
Element, the client's scalar and Element, and his random private
value, denoted here as server.private, he created as part of the
generation of his "commit" to compute the premaster secret as
follows:

```
z = F(scalar-op(server.private,
             element-op(ClientKeyExchange.Element,
                         scalar-op(ClientKeyExchange.scalar, PE))))
```

The intermediate value, z, is then used as the premaster secret after
any leading bytes of z that contain all zero bits have been stripped
off.

5. Ciphersuite Definition

    This memo adds the following ciphersuites:

        CipherSuite TLS_FFCPWD_WITH_3DES_EDE_CBC_SHA = (TBD, TBD);

        CipherSuite TLS_FFCPWD_WITH_AES_128_CBC_SHA = (TBD, TBD);

        CipherSuite TLS_ECCPWD_WITH_AES_128_CBC_SHA = (TBD, TBD);

        CipherSuite TLS_ECCPWD_WITH_AES_128_GCM_SHA256 = (TBD, TBD);

        CipherSuite TLS_ECCPWD_WITH_AES_256_GCM_SHA384 = (TBD, TBD);

        CipherSuite TLS_FFCPWD_WITH_AES_128_CCM_SHA = (TBD, TBD);

        CipherSuite TLS_ECCPWD_WITH_AES_128_CCM_SHA = (TBD, TBD);

        CipherSuite TLS_ECCPWD_WITH_AES_128_CCM_SHA256 = (TBD, TBD);

        CipherSuite TLS_ECCPWD_WITH_AES_256_CCM_SHA384 = (TBD, TBD);

    Implementations conforming to this specification MUST support the
    TLS_ECCPWD_WITH_AES_128_CCM_SHA ciphersuite; they SHOULD support
    TLS_FFCPWD_WITH_AES_128_CCM_SHA, TLS_FFCPWD_WITH_AES_128_CBC_SHA,

TLS_ECCPWD_WITH_AES_128_CBC_SHA, TLS_ECCPWD_WITH_AES_128_GCM_SHA256,
TLS_ECCPWD_WITH_AES_256_GCM_SHS384; and MAY support the remaining
ciphersuites.

When negotiated with a version of TLS prior to 1.2, the Pseudo-Random
Function (PRF) from that version is used; otherwise, the PRF is the
TLS PRF [RFC5246] using the hash function indicated by the
ciphersuite. Regardless of the TLS version, the TLS-PWD random
function, H, is always instantiated with the hash algorithm indicated
by the ciphersuite.

For those ciphersuites that use Cipher Block Chaining (CBC)
[SP800-38A] mode, the MAC is HMAC [RFC2104] with the hash function
indicated by the ciphersuite.

6. Acknowledgements

The authenticated key exchange defined here has also been defined for
use in 802.11 networks, as an EAP method, and as an authentication
method for IKE. Each of these specifications has elicited very
helpful comments from a wide collection of people that have allowed
the definition of the authenticated key exchange to be refined and
improved.

The authors would like to thank Scott Fluhrer for discovering the
"password as exponent" attack that was possible in an early version
of this key exchange and for his very helpful suggestions on the
techniques for fixing the PE to prevent it. The authors would also
like to thank Hideyuki Suzuki for his insight in discovering an
attack against a previous version of the underlying key exchange
protocol. Special thanks to Lily Chen for helpful discussions on
hashing into an elliptic curve and to Jin-Meng Ho for suggesting the
countermeasures to protect against a small sub-group attack. Rich
Davis suggested the defensive checks that are part of the processing
of the ServerKeyExchange and ClientKeyExchange messages, and his
various comments have greatly improved the quality of this memo and
the underlying key exchange on which it is based.

Martin Rex, Peter Gutmann, Marsh Ray, and Rene Struik, discussed the
possibility of a side-channel attack against the hunting-and-pecking
loop on the TLS mailing list. That discussion prompted the addition
of the security parameter, k, to the hunting-and-pecking loop.

7. IANA Considerations

IANA SHALL assign a value for a new TLS extention type from the TLS

ExtensionType Registry defined in [RFC5246] with the name "pwd". The
RFC editor SHALL replace TBD in Section 4.2.1 with the IANA-assigned
value for this extension.

IANA SHALL assign nine new ciphersuites from the TLS Ciphersuite
Registry defined in [RFC5246] for the following ciphersuites:

    CipherSuite TLS_FFCPWD_WITH_3DES_EDE_CBC_SHA = (TBD, TBD);

    CipherSuite TLS_FFCPWD_WITH_AES_128_CBC_SHA = (TBD, TBD);

    CipherSuite TLS_ECCPWD_WITH_AES_128_CBC_SHA = (TBD, TBD);

    CipherSuite TLS_ECCPWD_WITH_AES_128_GCM_SHA256 = (TBD, TBD);

    CipherSuite TLS_ECCPWD_WITH_AES_256_GCM_SHA384 = (TBD, TBD);

    CipherSuite TLS_FFCPWD_WITH_AES_128_CCM_SHA = (TBD, TBD);

    CipherSuite TLS_ECCPWD_WITH_AES_128_CCM_SHA = (TBD, TBD);

    CipherSuite TLS_ECCPWD_WITH_AES_128_CCM_SHA256 = (TBD, TBD);

    CipherSuite TLS_ECCPWD_WITH_AES_256_CCM_SHA384 = (TBD, TBD);

The RFC editor SHALL replace (TBD, TBD) in all the ciphersuites
defined in Section 5 with the appropriate IANA-assigned values.

8. Security Considerations

A passive attacker against this protocol will see the
ServerKeyExchange and the ClientKeyExchange containing the server's
scalar and Element, and the client's scalar and Element,
respectively. The client and server effectively hide their secret
private value by masking it modulo the order of the selected group.
If the order is "q", then there are approximately "q" distinct pairs
of numbers that will sum to the scalar values observed. It is
possible for an attacker to iterate through all such values but for a
large value of "q", this exhaustive search technique is
computationally infeasible. The attacker would have a better chance
in solving the discrete logarithm problem, which we have already
assumed (see Section 3.5) to be an intractable problem.

A passive attacker can take the Element from either the
ServerKeyExchange or the ClientKeyExchange and try to determine the
random "mask" value used in its construction and then recover the
other party's "private" value from the scalar in the same message.

But this requires the attacker to solve the discrete logarithm
problem which we assumed was intractable.

Both the client and the server obtain a shared secret, the premaster
secret, based on a secret group element and the private information
they contributed to the exchange. The secret group element is based
on the password. If they do not share the same password they will be
unable to derive the same secret group element and if they don't
generate the same secret group element they will be unable to
generate the same premaster secret. Seeing a finished message along
with the ServerKeyExchange and ClientKeyExchange will not provide any
additional advantage of attack since it is generated with the
unknowable premaster secret.

An active attacker impersonating the client can induce a server to
send a ServerKeyExchange containing the server's scalar and Element.
It can attempt to generate a ClientKeyExchange and send to the server
but the attacker is required to send a finished message first so the
only information she can obtain in this attack is less than the
information she can obtain from a passive attack, so this particular
active attack is not very fruitful.

An active attacker can impersonate the server and send a forged
ServerKeyExchange after receiving the ClientHello. The attacker then
waits until it receives the ClientKeyExchange and finished message
from the client. Now the attacker can attempt to run through all
possible values of the password, computing PE (see Section 4.1),
computing candidate premaster secrets (see Section 4.3), and
attempting to recreate the client's finished message.

But the attacker committed to a single guess of the password with her
forged ServerKeyExchange. That value was used by the client in her
computation of the premaster secret which was used to produce the
finished message. Any guess of the password which differs from the
one used in the forged ServerKeyExchange would result in each side
using a different PE in the computation of the premaster secret and
therefore the finished message cannot be verified as correct, even if
a subsequent guess, while running through all possible values, was
correct. The attacker gets one guess, and one guess only, per active
attack.

Instead of attempting to guess at the password, an attacker can
attempt to determine PE and then launch an attack. But PE is
determined by the output of the random function, H, which is
indistinguishable from a random source since H is assumed to be a
"random oracle" (Section 3.5). Therefore, each element of the finite
cyclic group will have an equal probability of being the PE. The
probability of guessing PE will be $1/q$, where q is the order of the

group. For a large value of "q" this will be computationally
infeasible.

The implications of resistance to dictionary attack are significant.
An implementation can provision a password in a practical and
realistic manner-- i.e. it MAY be a character string and it MAY be
relatively short-- and still maintain security. The nature of the
pool of potential passwords determines the size of the pool, D, and
countermeasures can prevent an attacker from determining the password
in the only possible way: repeated, active, guessing attacks. For
example, a simple four character string using lower-case English
characters, and assuming random selection of those characters, will
result in D of over four hundred thousand. An attacker would need to
mount over one hundred thousand active, guessing attacks (which will
easily be detected) before gaining any significant advantage in
determining the pre-shared key.

Countermeasures to deal with successive active, guessing attacks are
only possible by noticing a certain username is failing repeatedly
over a certain period of time. Attacks which attempt to find a
password for a random user are more difficult to detect. For
instance, if a device uses a serial number as a username and the pool
of potential passwords is sufficiently small, a more effective attack
would be to select a password and try all potential "users" to
disperse the attack and confound countermeasures. It is therefore
RECOMMENDED that implementations of TLS-pwd keep track of the total
number of failed authentications regardless of username in an effort
to detect and thwart this type of attack.

The benefits of resistance to dictionary attack can be lessened by a
client using the same passwords with multiple servers. An attacker
could re-direct a session from one server to the other if the
attacker knew that the intended server stored the same password for
the client as another server.

An adversary that has access to, and a considerable amount of control
over, a client or server could attempt to mount a side-channel attack
to determine the number of times it took for a certain password (plus
client random and server random) to select a password element. Each
such attack could result in a successive paring-down of the size of
the pool of potential passwords, resulting in a manageably small set
from which to launch a series of active attacks to determine the
password. A security parameter, k, is used to normalize the amount
of work necessary to determine the password element (see
Section 4.1). The probability that a password will require more than
k iterations is roughly $(q/2p)^k$ so it is possible to mitigate a side
channel attack at the expense of a constant cost per connection
attempt.

9. Implementation Considerations

   The selection of the ciphersuite and selection of the particular
   finite cyclic group to use with the ciphersuite are divorced in this
   memo but they remain intimately close.

   It is RECOMMENDED that implementations take note of the strength
   estimates of particular groups and to select a ciphersuite providing
   commensurate security with its hash and encryption algorithms. A
   ciphersuite whose encryption algorithm has a keylength less than the
   strength estimate, or whose hash algorithm has a blocksize that is
   less than twice the strength estimate SHOULD NOT be used.

   For example, the elliptic curve named secp256r1 (whose IANA-assigned
   number is 23) provides an estimated 128 bits of strength and would be
   compatible with an encryption algorithm supporting a key of that
   length, and a hash algorithm that has at least a 256-bit blocksize.
   Therefore, a suitable ciphersuite to use with secp256r1 could be
   TLS_ECCPWD_WITH_AES_128_GCM_SHA256.

   Resistance to dictionary attack means that the attacker must launch
   an active attack to make a single guess at the password. If the size
   of the pool from which the password was extracted was D, and each
   password in the pool 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 attacker to
   determine the password through repeated brute-force, active, guessing
   attacks. Implementations SHOULD take note of this fact and choose an
   appropriate pool of potential passwords-- i.e. make D big.
   Implementations SHOULD also take countermeasures, for instance
   refusing authentication attempts by a particular username 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.

10. References

10.1. Normative References

   [RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-
              Hashing for Message Authentication", RFC 2104,
              February 1997.

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

   [RFC3454] Hoffman, P. and M. Blanchet, "Preparation of
              Internationalized Strings ("stringprep")", RFC 3454,
              December 2002.

   [RFC4013] Zeilenga, K., "SASLprep: Stringprep Profile for User Names
              and Passwords", RFC 4013, February 2005.

   [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security
              (TLS) Protocol Version 1.2", RFC 5246, August 2008.

   [SP800-38A]
              National Institute of Standards and Technology,
              "Recommendation for Block Cipher Modes of Operation--
              Methods and Techniques", NIST Special Publication 800-38A,
              December 2001.

10.2. Informative References

   [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, 1993.

   [RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness
              Requirements for Security", BCP 106, RFC 4086, June 2005.

   [RFC4492] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B.
              Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites
              for Transport Layer Security (TLS)", RFC 4492, May 2006.

   [RFC5751] Ramsdell, B. and S. Turner, "Secure/Multipurpose Internet
              Mail Extensions (S/MIME) Version 3.2 Message
              Specification", RFC 5751, January 2010.

   [RFC5967] Turner, S., "The application/pkcs10 Media Type", RFC 5967,
              August 2010.

   [RFC6090] McGrew, D., Igoe, K., and M. Salter, "Fundamental Elliptic
              Curve Cryptography Algorithms", RFC 6090, February 2011.

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

Authors' Addresses

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

    Email: dharkins@arubanetworks.com

    Dave Halasz (editor)
    Halasz Ventures
    8401 Chagrin Road, Suite 10A
    Chagrin Falls, OH 44023
    United States of America

    Email: david.e.halasz@gmail.com