Transport Layer Security   D. Harkins, Ed.
Internet-Draft   Aruba Networks
Intended status: Standards Track   D. Halasz, Ed.
Expires: September 29, 2014   Halasz Ventures
                                                             March 28, 2014

Secure Password Ciphersuites for Transport Layer Security (TLS)
draft-ietf-tls-pwd-04

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.

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 September 29, 2014.

Copyright Notice

Copyright (c) 2014 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.

Table of Contents

1. Background 3
1.1. The Case for Certificate-less Authentication 3
1.2. Resistance to Dictionary Attack 3
2. Keyword Definitions 4
3. Introduction 4
3.1. Notation 4
3.2. Discrete Logarithm Cryptography 5
3.2.1. Elliptic Curve Cryptography 5
3.2.2. Finite Field Cryptography 6
3.3. Instantiating the Random Function 7
3.4. Passwords 8
3.5. Assumptions 8
4. Specification of the TLS-PWD Handshake 9
4.1. Protecting the Username 9
4.1.1. Construction of a Protected Username 10
4.1.2. Recovery of a Protected Username 11
4.2. Fixing the Password Element 12
4.2.1. Computing an ECC Password Element 14
4.2.2. Computing an FFC Password Element 16
4.3. Changes to Handshake Message Contents 17
4.3.1. Client Hello Changes 17
4.3.2. Server Key Exchange Changes 18
4.3.2.1. Generation of ServerKeyExchange 19
4.3.2.2. Processing of ServerKeyExchange 20
4.3.3. Client Key Exchange Changes 21
4.3.3.1. Generation of Client Key Exchange 21
4.3.3.2. Processing of Client Key Exchange 22
4.4. Computing the Premaster Secret 22
5. Ciphersuite Definition 23
6. Acknowledgements 23
7. IANA Considerations 24
8. Security Considerations 25
9. Implementation Considerations 28
10. References 29
10.1. Normative References 29
10.2. Informative References 29
Appendix A. Example Exchange 30
Authors' Addresses 34

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 like EST [RFC7030] 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.

len(x)
indicates the length in bits of the string x.

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

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

G.x
indicates the x-coordinate of a point, G, on an elliptic curve.

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).

TLS [RFC5246] allows for both FFC and ECC domain parameter sets to be conveyed verbosely by the server. This opens up the possibility of a malicious server offering a weak group, or one with a trapdoor, that would lead to a leaking of information during a run of the protocol. Therefore, if explicit domain parameter sets are used with TLS-PWD,

they MUST be agreed-upon a priori in an out-of-band fashion. Clients MUST NOT accept explicit domain parameter sets from a server that it has not previously agreed to accept.

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 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 the server's 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 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.

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].

If the server supports username protection (see Section 4.1), it is assumed that the server has chosen a domain parameter set and generated a username-protection keypair. The chosen domain parameter set and public key are assumed to be conveyed to the client at the time the client's username and password were provisioned.

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 "commitment" 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.

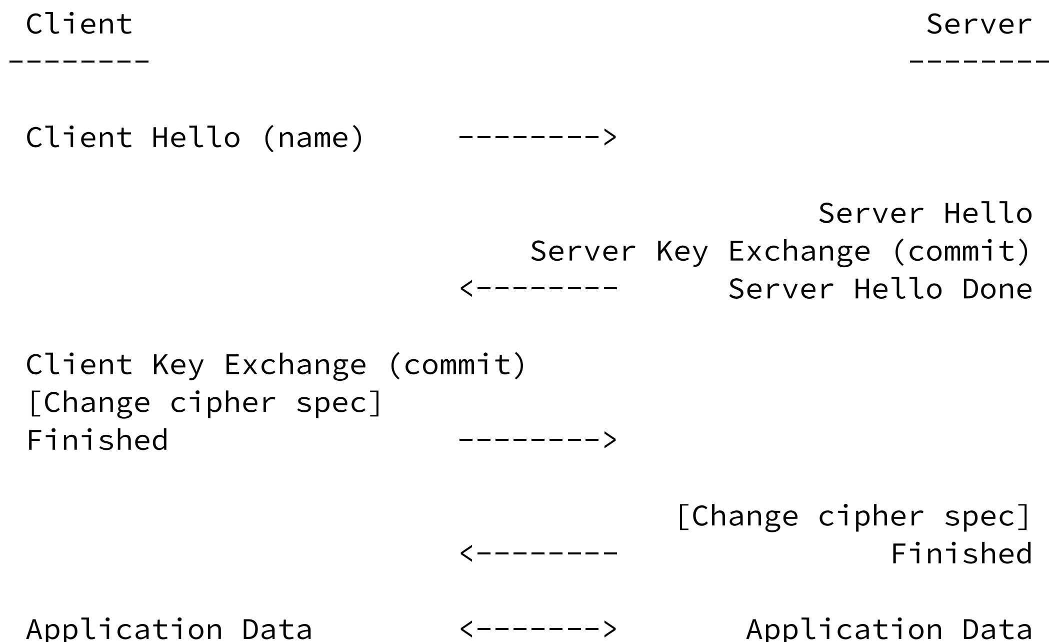

Figure 1

4.1. Protecting the Username

The client is required to identify herself to the server before the server can look up the appropriate client credential with which to

perform the authenticated key exchange. This has negative privacy implicaitons and opens up the client to tracking and increased

monitoring. It is therefore useful for the client to be able to protect her username from passive monitors of the exchange and against active attack by a malicious server. TLS-PWD provides such a mechsnism. Support for protected usernames is RECOMMENDED.

To enable username protection a server choses a domain parameter set, chooses a random private key, s, such that 1 < s < (q-1), where q is the order of the chosen group, uses scalar-op() with the selected group's generator to generate a public key, S:

    S = scalar-op(s, G)

This keypair SHALL only be used for username protection. For efficiency, the domain parameter set used for userame protection MUST be based on elliptic curve cryptography. Any ECC group that is approprate for TLS-PWD (see Section 3.2.1) is suitable for this purpose but for interoperability, brainpoolP256r1 MUST be supported. The domain parameter set used for username protection does not restrict the choice of domain parameter set used for the underlying key exchange in any way.

When the client's username and password are provisioned on the server, the server conveys the chosen group and its public key to the client. This is stored on the client along with the server-specific state (e.g. the hostname) it uses to initiate a TLS-PWD exchange. The server uses the same group and public key with all clients.

To protect a username, the client and server perform a static-ephemeral Diffie-Hellman exchange, using compact representation (and therefore compact output, see [RFC6090]). The result of the Diffie-Hellman exchange is passed to HKDF [RFC5869] to create a key-encrypting key suitable for AES-SIV [RFC5297]. The length of the key-encrypting key, l, and the hash function to use with HKDF depends on the length of the prime, p, of the group used to provide username protection:

o SHA-256, SIV-128, l=256 bits: when len(p) <= 256

o SHA-384, SIV-192, l=384 bits: when 256 < len(p) <= 384

o SHA-512, SIV-256, l=512 bits: when len(p) > 384

4.1.1. Construction of a Protected Username

Prior to initiating a TLS-PWD exchange, the client chooses a random secret, c, such that 1 < c < (q-1), where q is the order of the group from which the server's public key was generated, and uses scalar-op() with the group's generator to create a public key, C. It

uses scalar-op() with the server's public key and c to create a shared secret and derives a key-encrypting key, k, using the "salt-less" mode of HKDF [RFC5869].

```
C = scalar-op(c, G)

Z = scalar-op(c, S)

k = HKDF-expand(HKDF-extract(NULL, Z.x), "", l)
```

Where NULL indicates the salt-free invocation and "" indicates an empty string (i.e. there is no "context" passed to HKDF).

The key, k, and the client's username is then passed to SIV-encrypt with no AAD and no nonce to produce an encrypted username, u:

```
u = SIV-encrypt(k, username)
```

Note: the format of the ciphertext output from SIV includes the authenticating synthetic initialization vector.

The protected username SHALL be the concatenation of the x-coordinate of the client's public key, C, and the encrypted username, u. The length of the x-coordinate of C MUST be equal to the length of the group's prime, p, pre-pended with zeros, if necessary. The protected username is inserted into the PWD_name extension and the ExtensionType MUST be PWD_protect (see Section 4.3.1).

The length of the ciphertext output from SIV, minus the synthetic initialization vector, will be equal to the length of the input plaintext, in this case the username. To further foil traffic analysis, it is RECOMMENDED that clients append a series of NULL

bytes to their usernames prior to passing them to SIV-encrypt() and to vary the number of bytes added with each distinct run of TLS-PWD.

4.1.2. Recovery of a Protected Username

A server that receives a protected username needs to recover the client's username prior to performing the key exchange. To do so, the server computes the client's public key, completes the static-ephemeral Diffie-Hellman exchange, derives the key encrypting key, k, and decrypts the username.

The length of the x-coordinate of the client's public key is known (it is the length of the prime from the domain parameter set used to protect usernames) and can easily be separated from the ciphertext in the PWD_name extension in the Client Hello-- the first len(p) bits are the x-coordinate of the client's public key and the remaining

bits are the ciphertext.

Since compressed representation is used by the client, the server MUST compute the y-coordinate of the client's public key by using the equation of the curve:

```
y^2 = x^3 + ax + b
```

and solving for y. There are two solutions for y but since compressed output is also being used, the selection is irrelevant. The server reconstructs the client's public value, C, from (x, y). If there is no solution for y, or if (x, y) is not a valid point on the elliptic curve (see Section 3.2.1), the server MUST treat the Client Hello as if it did not have a password for a given username (see Section 4.3.1).

The server then uses scalar-op() with the reconstructed point C and the private key it uses for protected passwords, s, to generate a shared secret, and derives a key-encrypting key, k, in the same manner as in Section 4.1.1.

```
Z = scalar-op(s, C)

k = HKDF-expand(HKDF-extract(NULL, Z.x), "", l)
```

The key, k, and the ciphertext portion of the PWD_name extension, u, are passed to SIV-decrypt with no AAD and no nonce to produce the username:

    username = SIV-decrypt(k, u)

If SIV-decrypt returns the symbol FAIL indicating unsuccessful decryption and verification the server MUST treat the ClientHello as if it did not have a password for a given username (see Section 4.3.1). If successful, the server has obtained the client's username and can process it as needed. Any NULL octets added by the client prior to encryption can be easily stripped off of the string that represents the username.

4.2. Fixing the Password Element

Prior to making a "commitment" 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, m, is used to ensure that at least m iterations are always performed.

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, using the technique from section B.5.1 of [FIPS186-3], the pwd-seed is expanded using the PRF to the length of the prime from the

negotiated group's domain parameter set plus a constant sixty-four (64) to produce an intermediate pwd-tmp which is modularly reduced to create pwd-value:

```
n = len(p) + 64
pwd-tmp = PRF(pwd-seed, "TLS-PWD Hunting And Pecking",
              ClientHello.random | ServerHello.random) [0..n];
pwd-value = (pwd-tmp mod (p-1)) + 1
```

The pwd-value is then 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. 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, m.

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, m, SHOULD be set sufficiently large such that the probability that finding PE would take more than m iterations is sufficiently small (see Section 8).

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

4.2.1. 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 m iterations.

Algorithmically, the process looks like this:


```
found = 0
counter = 0
base = H(username | password | salt)
n = len(p) + 64
```

```
do {
  counter = counter + 1
  seed = H(base | counter | p)
  tmp = PRF(seed, "TLS-PWD Hunting And Pecking",
            ClientHello.random | ServerHello.random) [0..n]
  val = (tmp mod (p-1)) + 1
  if ( (val^3 + a*val + b) mod p is a quadratic residue)
    then
    if (found == 0)
    then
      x = val
      save = seed
      found = 1
      base = random()
    fi
  fi
} while ((found == 0) || (counter <= m))
y = sqrt(x^3 + a*x + b) mod p
if ( lsb(y) == lsb(save))
then
  PE = (x, y)
else
  PE = (x, p-y)
fi
```

Figure 2: Fixing PE for ECC Groups

Checking whether a value is a quadradic 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. 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 quadradic residue or a quadradic nonresidue 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()
} while ( lgr(qr, p) == -1)

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

Algorithmically, the masking technique to find out whether a value is a quadratic residue modulo a prime or not 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
}
```

The random quadratic residue and quadratic non-residue (qr and qnr above) can be used for all the hunting-and-pecking loops but the blinding value, r, MUST be chosen randomly for each loop.

4.2.2. 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
base = H(username | password | salt)
n = len(p) + 64
do {
  counter = counter + 1
  pwd-seed = H(base | counter | p)
  pwd-tmp = PRF(pwd-seed, "TLS-PWD Hunting And Pecking",
                ClientHello.random | ServerHello.random) [0..n]
  pwd-value = (pwd-tmp mod (p-1)) + 1
  PE = pwd-value ^ ((p-1)/q) mod p
  if (PE > 1)
  then
    found = 1
    base = random()
  fi
} while ((found == 0) || (counter <= m))
```

Figure 3: Fixing PE for FFC Groups

4.3. Changes to Handshake Message Contents

4.3.1. Client Hello Changes

The client is required to identify herself to the server by adding a either a PWD_protect or PWD_clear extension to the Client Hello message depending on whether the client wishes to protect its username (see Section 4.1) or not, respectively. The PWD_protect and PWD_clear extensions use 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. If username protection is performed, and the ExtensionType is PWD_protect, the contents of the PWD_name SHALL be constructed according to Section 4.1.1).

```
enum { PWD_clear(TBD1), PWD_protect(TBD2) } ExtensionType;

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

An unprotected PWD_name SHALL be UTF-8 encoded character string

processed according to the rules of the [RFC4013] profile of [RFC3454] and a protected PWD_name SHALL be a string of bits.

A client offering a PWD ciphersuite MUST include one of the PWD_name extensions in her Client Hello.

If a server does not have a password for a client identified by the username either extracted from the PWD_name, if unprotected, or

recovered using the technique in Section 4.1.2, if protected, or if recovery of a protected username fails, 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 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 4.2.2.

4.3.2. 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 "commitment" 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_selement<1..2^16-1>;
  opaque ff_sscalar<1..2^16-1>;
} ServerFFPWDParams;

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

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

4.3.2.1. Generation of ServerKeyExchange

The scalar and Element that comprise the server's "commitment" 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.

4.3.2.1.1. 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. If the client offered (an) elliptic curve(s) in its ClientHello using the Supported Elliptic Curves Extension, the server MUST include (one of the) named curve(s) in the ECParameters field in the ServerKeyExchange and the key exchange operations specified in Section 4.3.2.1 MUST use that group.

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.

4.3.2.1.2. 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.

4.3.2.2. Processing of ServerKeyExchange

Upon receipt of the ServerKeyExchange, the client decides whether to support the indicated group or not. If the client used the Supported Elliptic Curves Extension to offer (a) named curve(s) in her ClientHello, the named curve in the ServerKeyExchange MUST be one offered. If the server is explicitly specifying a group, either an FFC or ECC group, the client and server MUST have agreed upon groups prior to beginning the exchange (see Section 3.2) and the client MUST compare each field of the explicit offer to the agreed-upon group(s). Any discrepency SHALL result in the exchange being aborted.

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 and the server's "commitment" has been successfully validated, the client extracts the salt from the ServerKeyExchange and generates the password element, PE, according

to Section 3.4 and Section 4.2.2. If the group is not acceptable or the server's "commitment" failed validation, the eexchange MUST be aborted.

4.3.3. 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 "commitment" to the server. It, therefore, contains a scalar and an Element.

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

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

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

4.3.3.1. Generation of Client Key Exchange

The client's scalar and Element are generated in the manner described in Section 4.3.2.1.

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.

4.3.3.2. Processing of Client Key Exchange

Upon receipt of the ClientKeyExchange, the server must validate the client's "commitment" by ensuring that: 1) the client's scalar and element differ from the server's scalar and element; 2) the client's scalar value is greater than zero (0) and less than the order of the group, q; and 3) 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. If any of these three conditions are not met the server MUST abort the exchange.

4.4. Computing the Premaster Secret

The client uses 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 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_CBC_SHA ciphersuite; they SHOULD support TLS_ECCPWD_WITH_AES_128_CCM_SHA, TLS_FFCPWD_WITH_AES_128_CCM_SHA, TLS_ECCPWD_WITH_AES_128_GCM_SHA256, TLS_ECCPWD_WITH_AES_256_GCM_SHA384; 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. 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, m, to the hunting-and-pecking loop. Scott Flurer suggested the blinding technique to test whether a value is a quadratic residue modulo a prime in a manner that does not leak information about the value being tested.

7. IANA Considerations

IANA SHALL assign two values for a new TLS extention type from the TLS ExtensionType Registry defined in [RFC5246] with the name "pwd_protect" and "pwd_clear". The RFC editor SHALL replace TBD1 and TBD2 in Section 4.3.1 with the IANA-assigned value for these extensions.

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. The "DTLS-OK" column in the ciphersuite registry SHALL be set to "Y" for all ciphersuites defined in this memo.

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 4.2), computing candidate premaster secrets (see Section 4.4), 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, m, is used to normalize the amount of work necessary to determine the password element (see Section 4.2). The probability that a password will require more than m iterations is roughly (q/2p)^m for ECC groups and (q/p)^m for FFC groups, so it is possible to mitigate side channel attack at the expense of a constant cost per connection attempt. But if a particular password requires more than k iterations it will leak k bits of information to the side-channel attacker, which for some dictionaries will uniquely identify the password. Therefore, the security parameter, m, needs to be set with great care. It is RECOMMENDED that an implementation set the security parameter, m, 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 server uses a database of salted passwords. While this will prevent an adversary who gains access to the database from learning the client's password, it does not prevent such an adversary from impersonating the client back to the server. Each side uses the salted password, called the base, as the authenticaiton credential so the database of salted passwords MUST be afforded the security of a database of plaintext passwords.

Authentication is performed by proving knowledge of the password. Any third party that knows the password shared by the client and server can impersonate one to the other.

The static-ephemeral Diffie-Hellman exchange used to protect usernames requires the server to reuse its Diffie-Hellman public key. To prevent an invalid curve attack, an entity that reuses its Diffie-Hellman public key needs to check whether the received ephemeral public key is actually a point on the curve. This is done explicitly as part of the server's reconstruction of the client's public key out of only its x-coordinate ("compact representation").

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 brainpoolP256r1 (whose IANA-assigned number is 26) 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 brainpoolP256r1 could be TLS_ECCPWD_WITH_AES_128_GCM_SHA256 (see Appendix A for an example of such an exchange).

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.

[RFC5297] Harkins, D., "Synthetic Initialization Vector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES)", RFC 5297, October 2008.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, May 2010.

10.2. Informative References

[FIPS186-3] National Institute of Standards and Technology, "Digital Signature Standard (DSS)", Federal Information Processing Standards Publication 186-3.

[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.

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

[RFC7030] Pritikin, M., Yee, P., and D. Harkins, "Enrollment over Secure Transport", RFC 7030, October 2013.

[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.

[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.

Appendix A. Example Exchange

(Note: at the time of publication of this memo ciphersuites have not yet been assigned by IANA and the exchange that follows uses the private numberspace).

```
username: fred
password: barney

----  prior to running TLS-PWD ----

server generates salt:

96 3c 77 cd c1 3a 2a 8d 75 cd dd d1 e0 44 99 29
84 37 11 c2 1d 47 ce 6e 63 83 cd da 37 e4 7d a3

and a base:

6e 7c 79 82 1b 9f 8e 80 21 e9 e7 e8 26 e9 ed 28
c4 a1 8a ef c8 75 0c 72 6f 74 c7 09 61 d7 00 75

---- state derived during the TLS-PWD exchange ----

client and server agree to use brainpoolP256r1

client and server generate PE:
```

```
   PE.x:
   29 b2 38 55 81 9f 9c 3f c3 71 ba e2 84 f0 93 a3
   a4 fd 34 72 d4 bd 2e 9d f7 15 2d 22 ab 37 aa e6

   server private and mask:

   private:
   21 d9 9d 34 1c 97 97 b3 ae 72 df d2 89 97 1f 1b
   74 ce 9d e6 8a d4 b9 ab f5 48 88 d8 f6 c5 04 3c
   mask:
   0d 96 ab 62 4d 08 2c 71 25 5b e3 64 8d cd 30 3f
   6a b0 ca 61 a9 50 34 a5 53 e3 30 8d 1d 37 44 e5

   client private and mask:

   private:
   17 1d e8 ca a5 35 2d 36 ee 96 a3 99 79 b5 b7 2f
   a1 89 ae 7a 6a 09 c7 7f 7b 43 8a f1 6d f4 a8 8b
   mask:
   4f 74 5b df c2 95 d3 b3 84 29 f7 eb 30 25 a4 88
   83 72 8b 07 d8 86 05 c0 ee 20 23 16 a0 72 d1 bd

   both parties generate pre-master secret and master secret

   pre-master secret:
   01 f7 a7 bd 37 9d 71 61 79 eb 80 c5 49 83 45 11
   af 58 cb b6 dc 87 e0 18 1c 83 e7 01 e9 26 92 a4
   master secret:
   65 ce 15 50 ee ff 3d aa 2b f4 78 cb 84 29 88 a1
   60 26 a4 be f2 2b 3f ab 23 96 e9 8a 7e 05 a1 0f
   3d 8c ac 51 4d da 42 8d 94 be a9 23 89 18 4c ad

   ---- ssldump output of exchange ----

   New TCP connection #1: Charlene Client <-> Sammy Server
   1 1  0.0018 (0.0018)  C>SV3.3(173)  Handshake
         ClientHello
           Version 3.3
           random[32]=
             52 8f bf 52 17 5d e2 c8 69 84 5f db fa 83 44 f7
             d7 32 71 2e bf a6 79 d8 64 3c d3 1a 88 0e 04 3d
           cipher suites
           TLS_ECCPWD_WITH_AES_128_GCM_SHA256_PRIV
```

```
         TLS_ECCPWD_WITH_AES_256_GCM_SHA384_PRIV
         Unknown value 0xff
         compression methods
                   NULL
         extensions
```



```
         TLS-PWD unprotected name[5]=
           04 66 72 65 64
         elliptic curve point format[4]=
           03 00 01 02
         elliptic curve list[58]=
           00 38 00 0e 00 0d 00 1c 00 19 00 0b 00 0c 00 1b
           00 18 00 09 00 0a 00 1a 00 16 00 17 00 08 00 06
           00 07 00 14 00 15 00 04 00 05 00 12 00 13 00 01
           00 02 00 03 00 0f 00 10 00 11
   Packet data[178]=
     16 03 03 00 ad 01 00 00 a9 03 03 52 8f bf 52 17
     5d e2 c8 69 84 5f db fa 83 44 f7 d7 32 71 2e bf
     a6 79 d8 64 3c d3 1a 88 0e 04 3d 00 00 06 ff b3
     ff b4 00 ff 01 00 00 7a b8 aa 00 05 04 66 72 65
     64 00 0b 00 04 03 00 01 02 00 0a 00 3a 00 38 00
     0e 00 0d 00 1c 00 19 00 0b 00 0c 00 1b 00 18 00
     09 00 0a 00 1a 00 16 00 17 00 08 00 06 00 07 00
     14 00 15 00 04 00 05 00 12 00 13 00 01 00 02 00
     03 00 0f 00 10 00 11 00 0d 00 22 00 20 06 01 06
     02 06 03 05 01 05 02 05 03 04 01 04 02 04 03 03
     01 03 02 03 03 02 01 02 02 02 03 01 01 00 0f 00
     01 01


   1 2  0.0043 (0.0024)  S>CV3.3(94)  Handshake
          ServerHello
            Version 3.3
            random[32]=
              52 8f bf 52 43 78 a1 b1 3b 8d 2c bd 24 70 90 72
              13 69 f8 bf a3 ce eb 3c fc d8 5c bf cd d5 8e aa
            session_id[32]=
              ef ee 38 08 22 09 f2 c1 18 38 e2 30 33 61 e3 d6
              e6 00 6d 18 0e 09 f0 73 d5 21 20 cf 9f bf 62 88
            cipherSuite         TLS_ECCPWD_WITH_AES_128_GCM_SHA256_PRIV
            compressionMethod                   NULL
            extensions
```

```
          renegotiate[1]=
            00
          elliptic curve point format[4]=
            03 00 01 02
          heartbeat[1]=
            01
   Packet data[99]=
     16 03 03 00 5e 02 00 00 5a 03 03 52 8f bf 52 43
     78 a1 b1 3b 8d 2c bd 24 70 90 72 13 69 f8 bf a3
     ce eb 3c fc d8 5c bf cd d5 8e aa 20 ef ee 38 08
     22 09 f2 c1 18 38 e2 30 33 61 e3 d6 e6 00 6d 18
     0e 09 f0 73 d5 21 20 cf 9f bf 62 88 ff b3 00 00
```



```
     12 ff 01 00 01 00 00 0b 00 04 03 00 01 02 00 0f
     00 01 01

   1 3  0.0043 (0.0000)  S>CV3.3(141)  Handshake
        ServerKeyExchange
          params
            salt[32]=
              96 3c 77 cd c1 3a 2a 8d 75 cd dd d1 e0 44 99 29
              84 37 11 c2 1d 47 ce 6e 63 83 cd da 37 e4 7d a3
            EC parameters = 3
            curve id = 26
            element[65]=
              04 22 bb d5 6b 48 1d 7f a9 0c 35 e8 d4 2f cd 06
              61 8a 07 78 de 50 6b 1b c3 88 82 ab c7 31 32 ee
              f3 7f 02 e1 3b d5 44 ac c1 45 bd d8 06 45 0d 43
              be 34 b9 28 83 48 d0 3d 6c d9 83 24 87 b1 29 db
              e1
            scalar[32]=
              2f 70 48 96 69 9f c4 24 d3 ce c3 37 17 64 4f 5a
              df 7f 68 48 34 24 ee 51 49 2b b9 66 13 fc 49 21
   Packet data[146]=
     16 03 03 00 8d 0c 00 00 89 00 20 96 3c 77 cd c1
     3a 2a 8d 75 cd dd d1 e0 44 99 29 84 37 11 c2 1d
     47 ce 6e 63 83 cd da 37 e4 7d a3 03 00 1a 41 04
     22 bb d5 6b 48 1d 7f a9 0c 35 e8 d4 2f cd 06 61
     8a 07 78 de 50 6b 1b c3 88 82 ab c7 31 32 ee f3
     7f 02 e1 3b d5 44 ac c1 45 bd d8 06 45 0d 43 be
     34 b9 28 83 48 d0 3d 6c d9 83 24 87 b1 29 db e1
     00 20 2f 70 48 96 69 9f c4 24 d3 ce c3 37 17 64
```

```
     4f 5a df 7f 68 48 34 24 ee 51 49 2b b9 66 13 fc
     49 21


   1 4  0.0043 (0.0000)  S>CV3.3(4)  Handshake
         ServerHelloDone
   Packet data[9]=
     16 03 03 00 04 0e 00 00 00

   1 5  0.0086 (0.0043)  C>SV3.3(104)  Handshake
         ClientKeyExchange
           element[65]=
             04 a0 c6 9b 45 0b 85 ae e3 9f 64 6b 6e 64 d3 c1
             08 39 5f 4b a1 19 2d bf eb f0 de c5 b1 89 13 1f
             59 5d d4 ba cd bd d6 83 8d 92 19 fd 54 29 91 b2
             c0 b0 e4 c4 46 bf e5 8f 3c 03 39 f7 56 e8 9e fd
             a0
           scalar[32]=
             66 92 44 aa 67 cb 00 ea 72 c0 9b 84 a9 db 5b b8
```



```
             24 fc 39 82 42 8f cd 40 69 63 ae 08 0e 67 7a 48
   Packet data[109]=
     16 03 03 00 68 10 00 00 64 41 04 a0 c6 9b 45 0b
     85 ae e3 9f 64 6b 6e 64 d3 c1 08 39 5f 4b a1 19
     2d bf eb f0 de c5 b1 89 13 1f 59 5d d4 ba cd bd
     d6 83 8d 92 19 fd 54 29 91 b2 c0 b0 e4 c4 46 bf
     e5 8f 3c 03 39 f7 56 e8 9e fd a0 00 20 66 92 44
     aa 67 cb 00 ea 72 c0 9b 84 a9 db 5b b8 24 fc 39
     82 42 8f cd 40 69 63 ae 08 0e 67 7a 48

   1 6  0.0086 (0.0000)  C>SV3.3(1)  ChangeCipherSpec
   Packet data[6]=
     14 03 03 00 01 01


   1 7  0.0086 (0.0000)  C>SV3.3(40)  Handshake
   Packet data[45]=
     16 03 03 00 28 44 cd 3f 26 ed 64 9a 1b bb 07 c7
     0c 6d 3e 28 af e6 32 b1 17 29 49 a1 14 8e cb 7a
     0b 4b 70 f5 1f 39 c2 9c 7b 6c cc 57 20

   1 8  0.0105 (0.0018)  S>CV3.3(1)  ChangeCipherSpec
```

```
Packet data[6]=
  14 03 03 00 01 01


1 9  0.0105 (0.0000)  S>CV3.3(40)  Handshake
Packet data[45]=
  16 03 03 00 28 fd da 3c 9e 48 0a e7 99 ba 41 8c
  9f fd 47 c8 41 2c fd 22 10 77 3f 0f 78 54 5e 41
  a2 21 94 90 12 72 23 18 24 21 c3 60 a4


1 10 0.0107 (0.0002)  C>SV3.3(100)  application_data
Packet data....
```

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