Independent Submission
Request for Comments: 8492
Category: Informational
ISSN: 2070-1721

D. Harkins, Ed.
HP Enterprise
February 2019

Secure Password Ciphersuites for Transport Layer Security (TLS)

Abstract

This memo defines several new ciphersuites for the Transport Layer Security (TLS) protocol to support certificateless, secure authentication using only a simple, low-entropy password. The exchange is called "TLS-PWD". The ciphersuites are all based on an authentication and key exchange protocol, named "dragonfly", that is resistant to offline dictionary attacks.

Status of This Memo

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

This is a contribution to the RFC Series, independently of any other RFC stream. The RFC Editor has chosen to publish this document at its discretion and makes no statement about its value for implementation or deployment. Documents approved for publication by the RFC Editor are not candidates for any level of Internet Standard; see Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc8492.

Copyright Notice

Copyright (c) 2019 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 (https://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.

Table of Contents

1. Introduction and Motivation 3
   1.1. The Case for Certificateless Authentication 3
   1.2. Resistance to Dictionary Attacks 3
2. Key Words 4
3. Notation and Background 4
   3.1. Notation 4
   3.2. Discrete Logarithm Cryptography 5
        3.2.1. Elliptic Curve Cryptography 5
        3.2.2. Finite Field Cryptography 7
   3.3. Instantiating the Random Function 8
   3.4. Passwords 8
   3.5. Assumptions 9
4. Specification of the TLS-PWD Handshake 10
   4.1. TLS-PWD Pre-TLS 1.3 10
   4.2. TLS-PWD in TLS 1.3 11
   4.3. Protecting the Username 11
        4.3.1. Construction of a Protected Username 12
        4.3.2. Recovery of a Protected Username 13
   4.4. Fixing the Password Element 14
        4.4.1. Computing an ECC Password Element 16
        4.4.2. Computing an FFC Password Element 18
        4.4.3. Password Naming 19
        4.4.4. Generating TLS-PWD Commit 20
   4.5. Changes to Handshake Message Contents 20
        4.5.1. Pre-1.3 TLS 20
               4.5.1.1. ClientHello Changes 20
               4.5.1.2. ServerKeyExchange Changes 21
               4.5.1.3. ClientKeyExchange Changes 23
        4.5.2. TLS 1.3 24
               4.5.2.1. TLS 1.3 KeyShare 24
               4.5.2.2. ClientHello Changes 24
               4.5.2.3. ServerHello Changes 25
               4.5.2.4. HelloRetryRequest Changes 25
   4.6. Computing the Shared Secret 26
5. Ciphersuite Definition 26
6. IANA Considerations 27
7. Security Considerations 27
8. Human Rights Considerations 30
9. Implementation Considerations 31
10. References 32
   10.1. Normative References 32
   10.2. Informative References 33
Appendix A. Example Exchange 35
Acknowledgements 40
Author's Address 40

1. Introduction and Motivation

1.1. The Case for Certificateless Authentication

Transport Layer Security (TLS) usually uses public key certificates for authentication [RFC5246] [RFC8446]. 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 basic password exchange over the unauthenticated channel underscores this case.

- o The alternatives to TLS-PWD for employing certificateless TLS authentication -- using pre-shared keys in an exchange that is susceptible to dictionary attacks or using a Secure Remote Password (SRP) exchange that requires users to, a priori, be fixed to a specific Finite Field Cryptography (FFC) group for all subsequent connections -- are not acceptable for modern applications that require both security and cryptographic agility.

- 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 Enrollment over Secure Transport (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 cleartext. 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 Attacks

It is a common misconception that a protocol that authenticates with a shared and secret credential is resistant to dictionary attacks if the credential is assumed to be an N-bit uniformly random secret, where N is sufficiently large. The concept of resistance to dictionary attacks 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 attacks, 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. This means that the adversary will not be able to obtain any information about the password except whether a single guess from a single protocol run that 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 an attack and then go offline and 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. Key Words

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Notation and Background

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 FFC or Elliptic Curve Cryptography (ECC).

Elements in a group -- either an FFC or ECC group -- are indicated using uppercase, while scalar values are indicated using lowercase.

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]. Ciphersuites defined in this memo SHALL only use ECC curves based on the Weierstrass equation $y^2 = x^3 + a*x + b$.

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

- A prime, p, determining a prime field GF(p). The cryptographic group will be a subgroup of the full elliptic curve group, which consists of points on an elliptic curve -- Elements from GF(p) that satisfy the curve's equation -- together with the "point at infinity" that serves as the identity Element.

- Elements a and b from GF(p) that define the curve's equation. The point (x, y) in GF(p) x GF(p) is on the elliptic curve if and only if (y^2 - x^3 - a*x - b) mod p equals zero (0).

- A point, G, on the elliptic curve, which serves as a generator for the ECC group. G is chosen such that its order, with respect to elliptic curve addition, is a sufficiently large prime.

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

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

This memo uses the following ECC functions:

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

- 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 ECC Element, Z.

- 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".

- 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. Characteristic-2 curves SHALL NOT be used by 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 that 1) both coordinates x and y are greater than zero (0) and less than the prime defining the underlying field, 2) coordinates x and y satisfy the equation of the curve, and 3) they do not represent the "point at infinity". If any of those conditions are not true, the (x, y) pair is not a valid point on the curve.

A compliant implementation of TLS-PWD SHALL support group twenty-three (23) and SHOULD support group twenty-four (24) from the "TLS Supported Groups" registry; see [TLS_REG].

3.2.2. Finite Field Cryptography

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

- A prime, p, determining a prime field GF(p) (i.e., the integers modulo p). The FFC group will be a subgroup of GF(p)* (i.e., the multiplicative group of non-zero Elements in GF(p)).

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

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

This memo uses the following FFC functions:

- 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.

- 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.

- 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.

- 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 that 1) it is between one (1) and the prime, p, exclusive and 2) modular exponentiation of the integer by the group order, q, equals one (1). If either of these conditions is not true, the integer is not an Element in the group.

A compliant implementation of TLS-PWD SHOULD support group two hundred fifty-six (256) and group two hundred fifty-eight (258) from the "TLS Supported Groups" registry on [TLS_REG].

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 Hashed Message Authentication Code (HMAC) mode with a key whose length is equal to the 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, a password MAY be salted. When [RFC5246] or earlier is used, the password SHALL be salted. When [RFC8446] is used, a password MAY be stored with a salt or without. The password, username, and, optionally, the salt can create an irreversible digest called the "base", which is used in the authenticated key exchange.

The salting function is defined as:

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

The unsalted function is defined as:

```
base = SHA256(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 OpaqueString profile of [RFC8265], and the salt SHALL be a 32-octet random number. The server SHALL store a tuple of the form:

```
{ username, base, salt }
```

if the password is salted and:

```
{ username, base }
```

if it is not. When password salting is being used, the client generates the base upon receiving the salt from the server; otherwise, it may store the base at the time the username and password are provisioned.

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 entropy 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 4.3), 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 key exchange underlying TLS-PWD is the "dragonfly" password-authenticated key exchange (PAKE) as defined in [RFC7664].

The authenticated key exchange is accomplished by each side deriving a Password Element (PE) [RFC7664] in the chosen group, making a "commitment" to a single guess of the password using the PE, and generating a shared secret. The ability of each side to produce a valid finished message using a key derived from the shared secret allows each side to authenticates itself to the other side.

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

4.1. TLS-PWD Pre-TLS 1.3

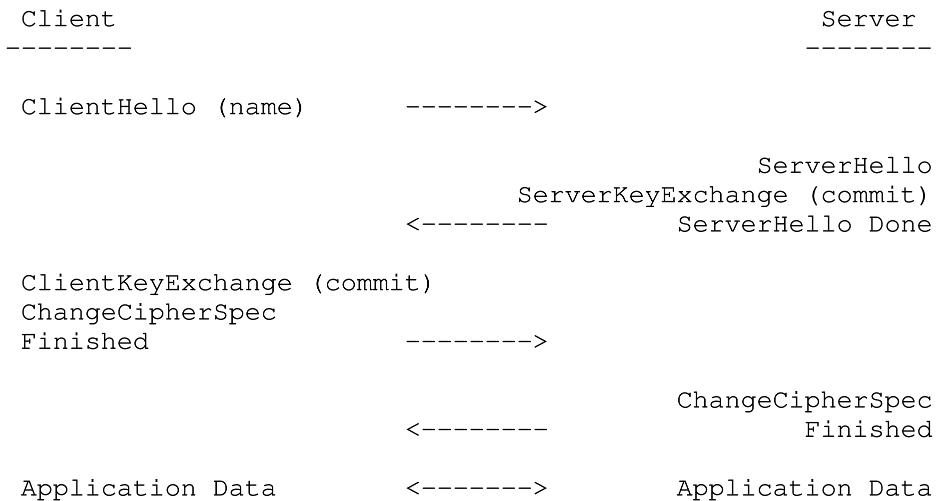

Figure 1: Pre-TLS 1.3 TLS-PWD Handshake

4.2. TLS-PWD in TLS 1.3

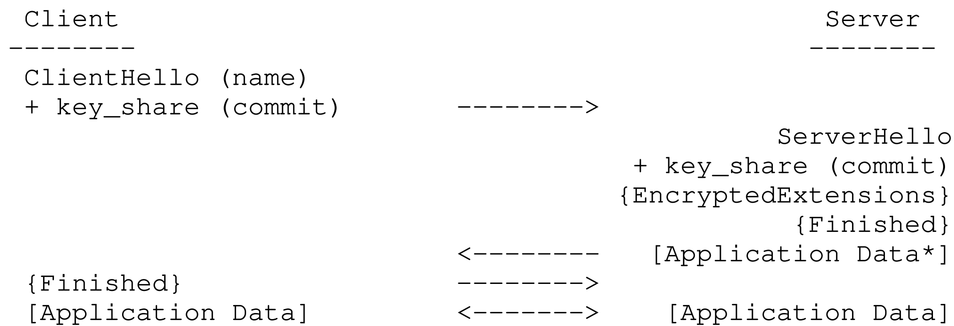

Figure 2: TLS 1.3 TLS-PWD Handshake

4.3. 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 implications 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 mechanism. Support for protected usernames is RECOMMENDED.

To enable username protection, a server chooses a domain parameter set and generates an ephemeral public/private keypair. This keypair SHALL only be used for username protection. For efficiency, the domain parameter set used for username protection MUST be based on ECC. Any ECC group that is appropriate for TLS-PWD (see Section 3.2.1) is suitable for this purpose, but for interoperability, prime256v1 (aka NIST's p256 curve) MUST be supported. The domain parameter set chosen for username protection is independent of the domain parameter set chosen for the underlying key exchange -- i.e., they need not be the same.

When the client's username and password are provisioned on the server, the chosen group and its public key are provisioned on 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. Since the y-coordinate is not necessary and eliminating it will reduce message size, compact representation (and therefore compact output; see [RFC6090]) is used

in the static-ephemeral Diffie-Hellman exchange. The result of the Diffie-Hellman exchange is passed to the HMAC-based Key Derivation Function (HKDF) [RFC5869] to create a key-encrypting key suitable for AES-SIV [RFC5297] (where "AES" stands for "Advanced Encryption Standard" and "SIV" stands for "Synthetic Initialization Vector") in its deterministic authenticated encryption mode. The length of the key-encrypting key (l) and the hash function to use with the HKDF depend on the length of the prime, p, of the group used to provide username protection:

- SHA-256, SIV-128, l=256 bits: when len(p) <= 256
- SHA-384, SIV-192, l=384 bits: when 256 < len(p) <= 384
- SHA-512, SIV-256, l=512 bits: when len(p) > 384

4.3.1. 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 it 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 it derives a key-encrypting key, k, using the "saltless" mode of the 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 the HKDF).

The client's username SHALL be represented as a UTF-8 encoded character string processed according to the rules of the OpaqueString profile of [RFC8265]. The output of OpaqueString is then passed with the key, k, to SIV-encrypt with no Additional Authenticated Data (AAD) and no nonce, to produce an encrypted username, u:

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

Note: The format of the ciphertext output includes the authenticating SIV.

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, prepended with zeros, if necessary. The protected username is inserted into the extension_data field of the pwd_protect extension (see Section 4.4.3).

To ensure that the username remains confidential, the random secret, c, MUST be generated from a source of random entropy; see Section 3.5.

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() such that the resulting padded length of the username is at least 128 octets.

4.3.2. 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 ClientHello -- 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 ClientHello as if it did not have a password for a given username (see Section 4.5.1.1).

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 it derives a key-encrypting key, k, in the same manner as that described in Section 4.3.1.

```
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 4.5.1.1). 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.

4.4. 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 the PE after it receives the ClientHello and chooses the particular group to use, and the client generates the PE prior to sending the ClientHello in TLS 1.3 and upon receipt of the ServerKeyExchange in TLS pre-1.3.

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

To thwart side-channel attacks that attempt to determine the number of iterations of the hunting-and-pecking loop that are used to find the 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 counter, the prime of the selected group, and the base (which is derived from the username, password, and, optionally, the salt; see Section 3.4):

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

Next, a context is generated consisting of random information. For versions of TLS less than 1.3, the context is a concatenation of the ClientHello random and the ServerHello random. For TLS 1.3, the context is the ClientHello random:

```
if (version < 1.3) {
  context = ClientHello.random | ServerHello.random
} else {
  context = ClientHello.random
}
```

Then, using the technique from Appendix B.5.1 of [FIPS186-4], the pwd-seed is expanded, using the Pseudorandom Function (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 the pwd-value:

```
n = len(p) + 64
pwd-tmp = PRF(pwd-seed, "TLS-PWD Hunting And Pecking",
              context) [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 PE or fails. If the group-specific operation fails, the counter is incremented, a new pwd-seed is generated, and the hunting-and-pecking process continues; this procedure continues until the group-specific operation returns the PE. After the PE has been chosen, the base is changed to a random number, the counter is incremented, and the hunting-and-pecking process 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 the PE would take more than m iterations is sufficiently small (see Section 7).

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

4.4.1. 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 the 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 the PE, a random number is used instead of the base, and the hunting-and-pecking process continues until it has looped through m iterations, where m is a suitably large number to prevent side-channel attacks (see [RFC7664]).

Algorithmically, the process looks like this:

```
found = 0
counter = 0
n = len(p) + 64
if (version < 1.3)
  context = ClientHello.random | ServerHello.random
} else {
  context = ClientHello.random
}
do {
  counter = counter + 1
  seed = H(base | counter | p)
  tmp = PRF(seed, "TLS-PWD Hunting And Pecking", context) [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 3: Fixing PE for ECC Groups

Checking whether a value is a quadratic residue modulo a prime can leak information about that value in a side-channel attack. Therefore, it is RECOMMENDED that the technique used to determine if the value is a quadratic residue modulo p first 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 quadratic residue or a quadratic 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 nonresidue (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.

4.4.2. Computing an FFC Password Element

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

the hunting-and-pecking process continues until it has looped through m iterations, where m is a suitably large number to prevent side-channel attacks (see [RFC7664]).

Algorithmically, the process looks like this:

```
found = 0
counter = 0
n = len(p) + 64
if (version < 1.3)
  context = ClientHello.random | ServerHello.random
} else {
  context = ClientHello.random
}
do {
  counter = counter + 1
  pwd-seed = H(base | counter | p)
  pwd-tmp = PRF(pwd-seed, "TLS-PWD Hunting And Pecking",
                context) [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 4: Fixing PE for FFC Groups

4.4.3. Password Naming

The client is required to identify herself to the server by adding either a pwd_protect or pwd_clear extension to her ClientHello message, depending on whether the client wishes to protect her username (see Section 4.3) or not, respectively. The pwd_protect and pwd_clear extensions use the standard mechanism defined in [RFC5246]. The "extension data" field of the 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 4.3.1.

```
enum { pwd_protect(29), pwd_clear(30) } ExtensionType;

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

An unprotected pwd_name SHALL be a UTF-8 encoded character string processed according to the rules of the OpaqueString profile of [RFC8265], and a protected pwd_name SHALL be a string of bits.

4.4.4. Generating TLS-PWD Commit

The scalar and Element that comprise each peer'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 (0) or one (1), the numbers must be thrown away and new random numbers generated. If their sum modulo the order of the group, q, is greater than one, 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 discussed 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.

4.5. Changes to Handshake Message Contents

4.5.1. Pre-1.3 TLS

4.5.1.1. ClientHello Changes

A client offering a PWD ciphersuite MUST include one of the pwd_name extensions from Section 4.4.3 in her ClientHello.

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 provided in Section 4.3.2 (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 ServerKeyExchange -- and then rejecting the client's finished message with a "bad_record_mac" alert [RFC8446]. To properly effect a simulated TLS-PWD exchange, an appropriate delay SHOULD be inserted between receipt of the ClientHello and response of the ServerHello. Alternately, a server

MAY choose to terminate the exchange if a password is not found. The security implication of terminating the exchange is to expose to an attacker whether a username is valid or not.

The server decides on a group to use with the named user (see Section 9) and generates the PE according to Section 4.4.2.

4.5.1.2. ServerKeyExchange Changes

The domain parameter set for the selected group MUST be explicitly specified by name in the ServerKeyExchange. ECC groups are specified using the NamedCurve enumeration of [RFC8422], and FFC groups are specified using the NamedGroup extensions added by [RFC7919] to the "TLS Supported Groups" registry in [TLS_REG]. 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 that was used to store the user's password.

Two new values have been added to the enumerated KeyExchangeAlgorithm to indicate TLS-PWD using FFC and TLS-PWD using ECC: ff_pwd and ec_pwd, respectively.

```
enum { ff_pwd, ec_pwd } KeyExchangeAlgorithm;

struct {
  opaque salt<1..2^8-1>;
  NamedGroup ff_group;
  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;
```

4.5.1.2.1. Generation of ServerKeyExchange

The scalar and Element referenced in this section are derived according to Section 4.4.4.

4.5.1.2.1.1. ECC ServerKeyExchange

ECC domain parameters are specified in the ECParameters component of the ECC-specific ServerKeyExchange as defined in [RFC8422]. 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 Supported Point Formats Extension (see [RFC8422]) in its ClientHello, 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 4.5.1.2.1 MUST use that group.

As mentioned in Section 3.2.1, characteristic-2 curves and curves with a co-factor greater than one (1) SHALL NOT be used by TLS-PWD.

4.5.1.2.1.2. FFC ServerKeyExchange

FFC domain parameters use the NamedGroup extension specified in [RFC7919]. 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.

4.5.1.2.2. Processing of ServerKeyExchange

Upon receipt of the ServerKeyExchange, the client decides whether to support the indicated group or not. 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 one (1) and less than the order of the group, q and 2) the Element is valid for the chosen group (see Sections 3.2.1 and 3.2.2 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 PE according to Sections 3.4 and 4.4.2. If the group is not acceptable or the server's "commitment" failed validation, the exchange MUST be aborted.

4.5.1.3. ClientKeyExchange 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;
```

4.5.1.3.1. Generation of ClientKeyExchange

The client's scalar and Element are generated in the manner described in Section 4.5.1.2.1.

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 Supported Point Formats Extension in its ClientHello, then the Element MUST be formatted in the ec_celement to conform to its initial request.

4.5.1.3.2. Processing of ClientKeyExchange

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 one (1) and less than the order of the group, q, and 3) the Element is valid for the chosen group (see Sections 3.2.1 and 3.2.2 for how to determine 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.

4.5.2. TLS 1.3

4.5.2.1. TLS 1.3 KeyShare

TLS 1.3 clients and servers convey their commit values in a "key_share" extension. The structure of this extension SHALL be:

```
enum { ff_pwd, ec_pwd } KeyExchangeAlgorithm;

struct {
    select (KeyExchangeAlgorithm) {
        case ec_pwd:
            opaque elemX[coordinate_length];
            opaque elemY[coordinate_length];
        case ff_pwd:
            opaque elem[coordinate_length];
     };
     opaque scalar<1..2^8-1>
} PWDKeyShareEntry;

struct {
     NamedGroup group;
     PWDKeyShareEntry pwd_key_exchange<1..2^16-1>;
} KeyShareEntry;
```

4.5.2.2. ClientHello Changes

The ClientHello message MUST include a pwd_name extension from Section 4.4.3 and it MUST include a key_share extension from Section 4.5.2.1.

Upon receipt of a ClientHello, the server MUST validate the key_share extension_data [RFC8446] to ensure that the scalar value is greater than one (1) and less than the order of the group q, and that the Element is valid for the chosen group (see Sections 3.2.1 and 3.2.2).

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 4.3.2 (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 its KeyShareEntry -- 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 ClientHello and response of the ServerHello. Alternately, a server MAY choose to terminate the exchange if a password is not found. The security implication of terminating the exchange is to expose to an attacker whether a username is valid or not.

4.5.2.3. ServerHello Changes

If the server supports TLS-PWD, agrees with the group chosen by the client, and finds an unsalted password indicated by the pwd_name extension of the received ClientHello, its ServerHello MUST contain a key_share extension from Section 4.5.2.1 in the same group as that chosen by the client.

Upon receipt of a ServerHello, the client MUST validate the key_share extension_data to ensure that the scalar value is greater than one (1) and less than the order of the group q, and that the Element is valid for the chosen group (see Sections 3.2.1 and 3.2.2).

4.5.2.4. HelloRetryRequest Changes

The server sends this message in response to a ClientHello if it desires a different group or if the password identified by the client's password identified by pwd_name is salted.

A different group is indicated by adding the KeyShareHelloRetryRequest extension to the HelloRetryRequest. The indication of a salted password, and the salt used, is done by adding the following structure:

```
enum { password_salt(31) } ExtensionType;

struct {
    opaque pwd_salt<2^16-1>;
} password_salt;
```

A client that receives a HelloRetryRequest indicating the password salt SHALL delete its computed PE and derive another version using the salt prior to sending another ClientHello.

4.6. Computing the Shared Secret

The client and server use their private value as calculated in Section 4.4.4 with the other party's Element and scalar for the ServerHello or ClientHello, respectively (here denoted "Peer_Element" and "peer_scalar") to generate the shared secret z.

```
z = F(scalar-op(private,
                elem-op(Peer_Element,
                        scalar-op(peer_scalar, PE))))
```

For TLS versions prior to 1.3, 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. For TLS version 1.3, leading zero bytes are retained, and the intermediate value z is used as the (EC)DHE input in the key schedule.

5. Ciphersuite Definition

This memo adds the following ciphersuites:

```
CipherSuite TLS_ECCPWD_WITH_AES_128_GCM_SHA256 = (0xC0,0xB0);

CipherSuite TLS_ECCPWD_WITH_AES_256_GCM_SHA384 = (0xC0,0xB1);

CipherSuite TLS_ECCPWD_WITH_AES_128_CCM_SHA256 = (0xC0,0xB2);

CipherSuite TLS_ECCPWD_WITH_AES_256_CCM_SHA384 = (0xC0,0xB3);
```

Implementations conforming to this specification MUST support the TLS_ECCPWD_WITH_AES_128_GCM_SHA256 ciphersuite; they SHOULD support the remaining ciphersuites.

When negotiated with a version of TLS prior to 1.2, the PRF from that earlier version is used; when the negotiated version of TLS is TLS 1.2, the PRF is the TLS 1.2 PRF [RFC5246], using the hash function indicated by the ciphersuite; when the negotiated version of TLS is TLS 1.3, the PRF is the Derive-Secret function from Section 7.1 of [RFC8446]. 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. IANA Considerations

IANA has assigned three values for new TLS extension types from the "TLS ExtensionType Values" registry defined in [RFC8446] and [RFC8447]. They are pwd_protect (29), pwd_clear (30), and password_salt (31). See Sections 4.5.1.1 and 4.5.2.2 for more information.

In summary, the following rows have been added to the "TLS ExtensionType Values" registry:

Value	Extension Name	TLS 1.3	Reference
29	pwd_protect	CH	RFC 8492
30	pwd_clear	CH	RFC 8492
31	password_salt	CH, SH, HRR	RFC 8492

IANA has assigned the following ciphersuites from the "TLS Cipher Suites" registry defined in [RFC8446] and [RFC8447]:

```
CipherSuite TLS_ECCPWD_WITH_AES_128_GCM_SHA256 = (0xC0,0xB0);

CipherSuite TLS_ECCPWD_WITH_AES_256_GCM_SHA384 = (0xC0,0xB1);

CipherSuite TLS_ECCPWD_WITH_AES_128_CCM_SHA256 = (0xC0,0xB2);

CipherSuite TLS_ECCPWD_WITH_AES_256_CCM_SHA384 = (0xC0,0xB3);
```

The "DTLS-OK" column in the registry has been set to "Y", and the "Recommended" column has been set to "N" for all ciphersuites defined in this memo.

7. Security Considerations

A security proof of this key exchange in the random oracle model is found in [lanskro].

A passive attacker against this protocol will see the ServerKeyExchange and the ClientKeyExchange (in TLS pre-1.3), or the KeyShare (from TLS 1.3), containing the scalar and Element of the server and the client, 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 the ServerKeyExchange or the ClientKeyExchange (in TLS pre-1.3), or from the KeyShare (from TLS 1.3), 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 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 shared secret. Seeing a finished message will not provide any additional advantage of attack, since it is generated with the unknowable secret.

In TLS pre-1.3, an active attacker impersonating the client can induce a server to send a ServerKeyExchange containing the server's scalar and Element. The attacker can attempt to generate a ClientKeyExchange and send it to the server, but she is required to send a finished message first; therefore, 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.

In TLS pre-1.3, 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 the PE (see Section 4.4), computing candidate premaster secrets (see Section 4.6), 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 that differs from the password used in the forged ServerKeyExchange would result in each side using a different PE in the computation of the premaster secret; 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 the PE and then launch an attack. But the 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 the 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 attacks 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 lowercase 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 that a certain username is failing repeatedly over a certain period of time. Attacks that 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 attacks can be lessened by a client using the same passwords with multiple servers. An attacker could redirect 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 PE. 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 PE (see Section 4.4). 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 attacks 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; for some dictionaries, this 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).

A database of salted passwords prevents 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 authentication 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").

8. Human Rights Considerations

At the time of publication of this document, there was a growing interest in considering the impacts that IETF (and IRTF) work can have on human rights; some related research is discussed in [RFC8280]. As such, the human rights considerations of TLS-PWD are presented here.

The key exchange underlying TLS-PWD uses public key cryptography to perform authentication and authenticated key exchange. The keys it produces can be used to establish secure connections between two people to protect their communication. Implementations of TLS-PWD, like implementations of other TLS ciphersuites that perform authentication and authenticated key establishment, are considered "armaments" or "munitions" by many governments around the world.

The most fundamental of human rights is the right to protect oneself. The right to keep and bear arms is an example of this right. Implementations of TLS-PWD can be used as arms, kept and borne, to defend oneself against all manner of attackers -- criminals, governments, lawyers, etc. TLS-PWD is a powerful tool in the promotion and defense of universal human rights.

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 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 block size 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) [RFC7027] provides an estimated 128 bits of strength and would be compatible with 1) an encryption algorithm supporting a key of that length and 2) a hash algorithm that has at least a 256-bit block size. 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 attacks 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 the 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, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/info/rfc2104>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/info/rfc5246>.

[RFC5297] Harkins, D., "Synthetic Initialization Vector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES)", RFC 5297, DOI 10.17487/RFC5297, October 2008, <https://www.rfc-editor.org/info/rfc5297>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

[RFC7919] Gillmor, D., "Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)", RFC 7919, DOI 10.17487/RFC7919, August 2016, <https://www.rfc-editor.org/info/rfc7919>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8265] Saint-Andre, P. and A. Melnikov, "Preparation, Enforcement, and Comparison of Internationalized Strings Representing Usernames and Passwords", RFC 8265, DOI 10.17487/RFC8265, October 2017, <https://www.rfc-editor.org/info/rfc8265>.

[RFC8422] Nir, Y., Josefsson, S., and M. Pegourie-Gonnard, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier", RFC 8422, DOI 10.17487/RFC8422, August 2018, <https://www.rfc-editor.org/info/rfc8422>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[RFC8447] Salowey, J. and S. Turner, "IANA Registry Updates for TLS and DTLS", RFC 8447, DOI 10.17487/RFC8447, August 2018, <https://www.rfc-editor.org/info/rfc8447>.

[TLS_REG] IANA, "Transport Layer Security (TLS) Parameters", <https://www.iana.org/assignments/tls-parameters/>.

10.2. Informative References

[FIPS186-4] National Institute of Standards and Technology, "Digital Signature Standard (DSS)", Federal Information Processing Standards Publication 186-4, DOI 10.6028/NIST.FIPS.186-4, July 2013, <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>.

[lanskro] Lancrenon, J. and M. Skrobot, "On the Provable Security of the Dragonfly Protocol", ISC 2015 Proceedings of the 18th International Conference on Information Security - Volume 9290, pp. 244-261, DOI 10.1007/978-3-319-23318-5_14, September 2015.

[RANDOR] Bellare, M. and P. Rogaway, "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols", Proceedings of the 1st ACM Conference on Computer and Communications Security, pp. 62-73, ACM Press, DOI 10.1145/168588.168596, November 1993.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

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

[RFC7027] Merkle, J. and M. Lochter, "Elliptic Curve Cryptography (ECC) Brainpool Curves for Transport Layer Security (TLS)", RFC 7027, DOI 10.17487/RFC7027, October 2013, <https://www.rfc-editor.org/info/rfc7027>.

[RFC7030] Pritikin, M., Ed., Yee, P., Ed., and D. Harkins, Ed., "Enrollment over Secure Transport", RFC 7030, DOI 10.17487/RFC7030, October 2013, <https://www.rfc-editor.org/info/rfc7030>.

[RFC7664] Harkins, D., Ed., "Dragonfly Key Exchange", RFC 7664, DOI 10.17487/RFC7664, November 2015, <https://www.rfc-editor.org/info/rfc7664>.

[RFC8280] ten Oever, N. and C. Cath, "Research into Human Rights Protocol Considerations", RFC 8280, DOI 10.17487/RFC8280, October 2017, <https://www.rfc-editor.org/info/rfc8280>.

[SP800-38A] Dworkin, M., "Recommendation for Block Cipher Modes of Operation - Methods and Techniques", NIST Special Publication 800-38A, DOI 10.6028/NIST.SP.800-38A, December 2001, <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf>.

[SP800-56A] Barker, E., Chen, L., Roginsky, A., Vassilev, A., and R. Davis, "Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography", NIST Special Publication 800-56A, Revision 3, DOI 10.6028/NIST.SP.800-56Ar3, April 2018, <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar3.pdf>.

Appendix A. Example Exchange

```
   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 the 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 premaster secret and master secret

premaster 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
        ciphersuites
        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....
```

Acknowledgements

The authenticated key exchange defined here has also been defined for use in 802.11 networks, as an Extensible Authentication Protocol (EAP) method, and as an authentication method for the Internet Key Exchange Protocol (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 author 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 author 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 on the TLS mailing list the possibility of a side-channel attack against the hunting-and-pecking loop. That discussion prompted the addition of the security parameter, m, to the hunting-and-pecking loop. Scott Fluhrer 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.

Author's Address

Dan Harkins (editor)
HP Enterprise
3333 Scott Blvd.
Santa Clara, CA 95054
United States of America

Email: dharkins@lounge.org