Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 21, 2016

D. Kuegler
BSI
J. Schmidt
secunet Security Networks
October 19, 2015

Using Password-Authenticated Key Agreement (PAKE) schemes in TLS
draft-schmidt-pake-tls-00

Abstract

This document describes how to integrate Password-Authenticated Key Agreement (PAKE) schemes into TLS. These schemes enable two parties who share a potentially weak password to derive a common cryptographic key, allowing them to establish a secure channel. The current document defines a generic way to integrate PAKE schemes into TLS. In addition, it demonstrates how to use the well-known Password Authenticated Connection Establishment (PACE) scheme in TLS as an example for the generic construction.

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 April 21, 2016.

Copyright Notice

Table of Contents

1. Requirements notation

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 [RFC2119].

2. Introduction

Assume two or more parties want to communicate in a protected manner. Using a shared password might be a simple possibility, since it is easy to remember and to transfer, e.g. via phone. However, it does

not deliver a strong security, in particular not the level that is provided by a cryptographic key. Password-Authenticated Key Agreement (PAKE) schemes provide a combination of both benefits. They allow the parties to derive a cryptographic key from a password to establish a secure communication. A PAKE scheme uses the password in a way that prevents an adversary from performing a brute-force attack without interaction with a legitimate communication partner. This way, an adversary trying to iterate through all possible passwords is detected and hence such attacks are mitigated. This memo shows how to integrate PAKE schemes into TLS.

As an example of how to use PAKE in TLS, it is shown how to integrate the Password Authenticated Connection Establishment (PACE), a realization of such a PAKE. It is balanced in the sense that both parties possess the same representation of the shared password. It comes with a security proof based on number-theoretic assumptions related to the Diffie-Hellman problem using random oracles and ideal ciphers.

3. PAKE Specification

This section gives a definitions and preliminaries required for PACE and describes the scheme itself.

3.1. Requirements

3.1.1. Notation

The following notation is used in this memo:

A||B: Concatenation of the values A and B

3.1.2. Group Specification

Most PAKE schemes rely on finite groups. Here, we consider two different groups: (1) groups over finite fields and (2) groups over elliptic curves. They have in common that, using appropriate parameters, the discrete logarithm problem in both groups is hard to solve.

- Let p be a prime for which the prime factors of (p-1)/2 are sufficiently large. Let g be a generator of the subgroup of GF(p) with q elements, whereas q is prime.

- For elliptic curve cryptography (ECC) this document relies on the definitions and algorithms described in [RFC6090]. In particular, the domain parameters used in this document are a prime p defining the field GF(p). Two elements a,b of GF(p) give the curve

equation. All points (x,y) in GF(p)^2 that satisfy the equation y^2=x^3+a*x+b mod p together with the point at infinity build the additive group of the curve, in which the computations are taking place. Furthermore, let G be an element of the EC group that generates a sub-group having a sufficient large prime order q. Over this group, we define the addition of two elements X,Y denoted as X+Y as well as a scalar multiplication k*G as adding k times the point G. Note that irrespective of the description of the curve in Weierstrass form here, other curves can be used as well.

In the following, if the expressions are valid for both groups, we will use the multiplicative notation, i.e. a*b and a^c, which maps to a+b and c*a for the ECC case.

3.1.3. Functions

In order to describe PACE, this memo makes use of the following cryptographic primitives:

- A symmetric block cipher (E,D) that encrypts a message M using a key K with E(K,M) to the chipertext C and decrypts C with D(K,C).

- A hash-function (H) that maps an arbitrary input X to a fixed-sized output Y=H(X).

- A MAC-function (MAC) that generates an authentication tag T for the message M using the key K by T=MAC(K,M).

Note that the algorithms that realized these primitives are given together with the identifier of the used PAKE scheme in our construction.

In addition, a function Hash2Point that maps from a random bit string to a point, either in the finite field or on the elliptic curve, is chosen. How this function operates depends on the used parameters:

Finite Field: In order to map a bit string S to a point m in a finite field GF(p), its hash is computed. This hash is mapped into a group of size q: m = (H(S))^((p-1)/q) mod p. Note that it MUST be verified, that m > 1 holds.

Weierstrass Curve: Weierstrass curves with the equation y^2=x^3+a*x+b mod p can use the map defined by Shallue, Woestijne [SW2006] and Ulas [U2007]. They showed that, defining g(u) = x^3+a*x+b, for at least one of the values x1(t,u)=u, x2(t,u)=-(b/a)*(1+1/(t^2*g(u)^2+t*g(u))) and x3(t,u)=t^3*g(u)^3*x2(t,u) exists an y with g(xi) = y^2 mod p. Based on this map, the Hash2Point

function is defined. Let u be the smallest value such that no y with y^2=g(u) exists. Values of u and g(u) for different curves are given in Table 1. Let sqrt(x) denote the square root of the element x. In case p= 3 mod 4, sqrt(x)=x^((p+1)/4). The Hash2Point function for a string S is defined as Hash2Point(S) = (x2(H(S),u),sqrt(g(x2(H(S),u))) if g(x2(H(S),u))^((p+1)/2) =g(x2(H(S),u))^2 else (x3(H(S),u),sqrt(g(x3(H(S),u)))). Note that in order to achieve a constant time implementation, both points have to be computed.

Curve25519: For curves y^2=x^3+a*x^2+b*x mod p with a*b*(a^2-4b) not null, Elligator2 [BHKL2013] as defined by Bernstein et al. is used as Hash2Point function. Let |y|=y if 0<y<(p-1)/2 and |y|=-y otherwise. In order to compute the map, a square root function, sqrt(x), is defined. Since p=5 mod 8 holds for Curve25519, let h = x^(p+3)/8, sqrt(x) = |h| if b^4=a^2 and sqrt(x)=|sqrt(-1)*h| otherwise. A bit string S is first mapped to GF(p), the underlying prime field by m = H(S) mod p. Subsequently, v=-a/(1+2*m^2), e=(v^3+a*v^2+b*v)^((q-1)/2), x =e*v-(1-e)*a/2 mod p, y = -e*sqrt(x^3+a*x^2+b*x) mod p are computed. The result of Hash2Point(S) is P=(x,y).

Name	u	g(u)
secp256r1	1	0x5ac635d8aa3a93e7 b3ebbd55769886bc 651d06b0cc53b0f6 3bce3c3e27d26049
secp384r1	1	0xb3312fa7e23ee7e4 988e056be3f82d19 181d9c6efe814112 0314088f5013875a c656398d8a2ed19d 2a85c8edd3ec2aed
secp521r1	3	0x51953eb9618e1c9a 1f929a21a0b68540 eea2da725b99b315 f3b8b489918ef109 e156193951ec7e93 7b1652c0bd3bb1bf 073573df883d2c34 f1ef451fd46b503f 12
brainpoolP256r1	0	0x26dc5c6ce94a4b44 f330b5d9bbd77cbf 958416295cf7e1ce 6bccdc18ff8c07b6
brainpoolP384r1	0	0x04a8c7dd22ce2826 8b39b55416f0447c 2fb77de107dcd2a6 2e880ea53eeb62d5 7cb4390295dbc994 3ab78696fa504c11
brainpoolP512r1	0	0x3df91610a83441ca ea9863bc2ded5d5a a8253aa10a2ef1c9 8b9ac8b57f1117a7 2bf2c7b9e7c1ac4d 77fc94cadc083e67 984050b75ebae5dd 2809bd638016f723

Table 1: Values of u and g(u) for different Weierstrass curves.

3.2. PACE

PACE is a balanced PAKE scheme that is used for e.g. travel documents [BFK2009]. In a nutshell, a nonce is generated by the initiating party and send, encrypted with the (hash of) the shared password, to the second party. The possession of the correct password is proven by the ability to encrypt and decrypt the nonce. The parties generate jointly, based on the nonce, a group for a Diffie-Hellman key agreement. Finally, the encryption and authentication keys are derived from the Diffie-Hellman key.

The precondition for the execution of PACE is that both parties share a password. If the domain parameters are not agreed beforehand, the initializing party can send (a set of) supported parameter together with the nonce in the first message, the choice can be done by the second party by reflecting the parameters to use. Hence, we assume that Alice and Bob share a finite group, i.e. a finite field or

domain parameters for an elliptic curve, its generator g with order q. All following computations are performed in this group.

A->B: Alice selects a random integer s with a fixed bit-length (in practice it has to be a multiple of the block-length of the symmetric cipher) and encrypts it using the shared password PW. Alice sends E(PW,s) together with the domain parameters to Bob.

B->A: Bob decrypts s, selects a random integer t and derives g' = Hash2Point(s||t). Moreover, Bob chooses y_b<q and computes Y_b = (g')^y_b. Bob sends t and Y_b to Alice.

A->B: Alice verifies that Y_b is a valid not-null element of the specified group, and derives g' = Hash2Point(s||t). Alice chooses y_a<q and computes Y_a=(g')^y_a. Afterwards, Alice derives K=Y_b^y_a and computes an encryption key K_enc= H(K||1), a mac key K_mac(K||2), as well as MAC key for the verification tag K_tag=H(K||3). Alice computes T_a = MAC(K_tag,(Y_b||g')) and sends Y_a and T_a to Bob.

B->A: Bob verifies that Y_a is a valid element in the group and especially is not null. Using Y_a and y_b, Bob computes the shared key K=Y_a^y_b and derives the encryption key K_enc= H(K||1), the mac key K_mac(K||2), as well as MAC key for the verification tag K_tag=H(K||3). Using these keys, Bob verifies T_a. If T_a is valid, Bob computes T_b=MAC(K_tag,(Y_a||g')) and sends T_b to Alice.

A: Alice verifies T_b. If T_b is valid, Alice and Bob can use the shared keys K_enc and K_mac to establish a secured channel.

After performing this protocol, both parties share a secret key that can be used for further communication.

3.2.1. Performance

Key agreement using PACE requires exchanging 2 messages from Alice to Bob and 2 responses. This makes PACE a 2-round protocol. The sizes of the messages are defined by the domain parameters: The first message contains the encrypted nonce and possibly domain parameters. Bobs response contains a group element. Alice and Bob require the selection of 2 random variables, each. Both perform 2 exponentiations and one evaluation of Hash2Point.

3.3. Handling PAKE Passwords

The passwords that are used in the PAKE schemes can be stored plain or protected either by hashing them or even using a salt with the hash. Furthermore, in case an augmented PAKE scheme is used, the representation of the password of the client and the server differs.

4. PAKE Integration in TLS

In order to integrate PAKE schemes into TLS, the Hello Messages and the KeyExchange Messages have to be adapted. The client sends an additional extension with the ClientHello message in order to initiate a key-exchange using a PAKE. Furthermore, the extension may contain details to the desired PAKE like the identity of the client, the desired group parameters, and potential nonces. We show how to integrate a PAKE scheme using the example of PACE.

4.1. Changes in ClientHello

The client notifies the Server with the ClientHallo messages that a PAKE scheme should be used for the key-agreement and sends a list of supported schemes. This is done by an extension of the Hello message. Moreover, the extension contains the information on the identity of the client and furthermore which set of schemes is supported.

```
enum{PAKE(TBD1)} ExtensionType;
    struct{
        opaque ClientName <1..2^8-1>
        enum{PACE_AES128_SHA256(TBD2),
             PACE_ECC_AES128_SHA256(TBD3),
            } PAKEType;
        } PAKEParams;
```

The parameter ClientName in the extension is used by the server to identify the client and determine which password to use for further messages. The ClientName SHALL be a UTF-8 encoded character string processed according to the [RFC4013] profile of [RFC3454]. The PAKEType parameter defines which PAKE is supported by the client.

In case all requested PAKEs are unknown to the server or the requested domain parameters do not fit the expected ones, the server shall respond with an unknown cipher error. In case the supplied ClientName is unknown, the server should keep running the protocol using random elements that fit the expected answers and reject the TLS clients finished by sending a bad_record_mac alert.

4.1.1. ClientHello for PACE_AES128_SHA256 / PACE_ECC_AES128_SHA256

The PAKEType PACE_AES128_SHA256 / PACE_ECC_AES128_SHA256 defines PACE using a finite field / an elliptic curve. As block cipher, AES128 is used, as MAC function an HMAC with SHA256.

Note that the client indicates whether a finite field, an elliptic curve or both can be used for PACE. In case ECC is supported, the client can further specify the supported curves and the point format by the Supported Elliptic Curves and Supported Point Formats extensions as defined in [RFC4492]. The selection of the used parameter set is done by the server.

4.2. Changes in ServerKeyExchange

The ServerKeyExchange message contains the response of the server to the PAKE request in the ClientHello message. The response should reflect one of the PAKEs supported by the client and send an appropriate answer back to the client.

```
enum{PACE_AES128_SHA256(TBD2),
        PACE_ECC_AES128_SHA256(TBD3)} KeyExchangeAlgorithm;

struct {
    opaque p<1..2^16-1>
    opaque g<1..2^16-1>
    opaque q<1..2^16-1>
    opaque t[16];
    opaque Y_s<1..2^16-1>
}PACE_parameters

struct {
    ECParameters elliptic_curve;
    opaque t[16];
    ECPoint Y_s;
}PACE_ECC_parameters

struct {
    select (KeyExchangeAlgorithm)
    case PACE_AES128_SHA256:
        opaque PACE_parameters;
    case PACE_ECC_AES128_SHA256:
        opaque PACE_ECC_parameters;
} ServerKeyExchange;
```

4.2.1. ServerKeyExchange for PACE_AES128_SHA256

Upon reception of the ClientHello Message with PAKEType PACE_AES128_SHA256, the server selects appropriate domain parameters, i.e. a prime p and a generator g with order q. The server looks up the password corresponding to the ClientName sent and uses it to decrypt the first 16 bytes of the ClientRandom variable to s. Note that the ClientRandom is interpreted an encrypted value even though it was randomly selected by the client. This spares the parties from exchanging an additional nonce. The server furthermore selects two random numbers t < 2^128 and y_s < q and computes g'= Hash2Point(s||t) and verifies g' > 1. Otherwise, the server repeats the selection of t. The server computes Y_s=(g')^y_s mod p and answers by reflecting PACE_AES128_SHA256 as KeyExchangeAlgorithm and sending the values p, g, t and Y_s to the client.

4.2.2. ServerKeyExchange for PACE_ECC_AES128_SHA256

Upon reception of the ClientHello Message with PAKEType PACE_ECC_AES128_SHA256, the server selects appropriate domain parameters. If the client sent the Supported Elliptic Curves and/or Supported Point Formats extension, the server MUST select one of the supported parameters. The server looks up the password corresponding to the ClientName sent and uses it to decrypt the first 16 bytes of the ClientRandom variable to s. Note that the ClientRandom is interpreted an encrypted value even though it was randomly selected by the client. This spares the parties from exchanging an additional nonce. The server furthermore selects two random numbers t<2^128 and y_s<q, computes G'= Hash2Point(s||t) and verifies that G' is not in a small subgroup for curves with cofactor > 1. In case this check fails, the server repeats the selection of t. Afterwards, the server computes Y_s=y_s*G'. The server answers by reflecting PACE_ECC_AES128_SHA256 as KeyExchangeAlgorithm and sending the domain parameters together with t and Y_s, encoded a ECPoint as defined in [RFC4492] to the client.

4.3. Changes in ClientKeyExchange

The ClientKeyExchange message finalizes the key exchange.

```
struct {
    select (PAKEType)
            case PACE_AES128_SHA256:  opaque Y_c<1..2^16-1>
            case PACE_ECC_AES128_SHA256: ECPoint Y_c;
} ClientKeyPAKEParams;
```

4.3.1. ClientKeyExchange for PACE_AES128_SHA256

Upon reception of the ServerKeyExchange Message with KeyExchangeAlgorithm PACE_AES128_SHA256, the client verifies the received parameters. In particular, it verifies that the domain parameters are valid and that Y_s is a not-null element of GF(p). After successful verification, the client uses the shared password to derive s from the first 16 Bytes of the ClientRandom value sent to the server in the ClientHello message. The client selects a random number y_c < q, computes g'= Hash2Point(s||t), verifies g' > 1 and computes Y_c = g'^y_c mod p as well as K = Y_s^y_c mod p. In case g' = 0 or g' = 1, the client SHALL NOT immediately report an error but use a random g' for all further computations and reject the server's Finished message. The client send Y_c to the server, which also derives K=Y_c^y_s mod p after validation of Y_c. Both parties now use the shared value K to derive an encryption key K_enc = H(K||1), a MAC key K_mac=H(K||2), as well as MAC key for the verification tag K_tag=H(K||3). The validation tag of the client is T_c=MAC(K_tag,"PACE_AES128_SHA256"||Y_c) and for the server T_s=MAC(K_tag,"PACE_AES128_SHA256"||Y_s).

4.3.2. ClientKeyExchange for PACE_ECC_AES128_SHA256

Upon reception of the ServerKeyExchange Message with KeyExchangeAlgorithm PACE_ECC_AES128_SHA256, the client verifies the received parameters. In particular, it verifies the domain parameters and that Y_s is an element of the specified curve and is not equal to the point at infinity. After successful verification, the client uses the shared password to derive s from the first 16 Bytes of the ClientRandom value sent to the server in the ClientHello message. The client computes G'= Hash2Point(s||t) and verifies that G' is not in a small subgroup for curves with cofactor > 1. In case this check fails, the client SHALL NOT immediately report an error but use a random G' for all further computations and reject the server's Finished message. After successful verification, the client selects a random number y_c<q and and computes K=y_c*Y_s. The client sends Y_c, encoded a ECPoint as defined in [RFC4492], to the server, which also derives K=y_s*Y_c after validation of Y_c. Both parties now use the x-coordinate of the shared value K denoted as K_x to derive an encryption key K_enc = H(K_x||1), a MAC key K_mac=H(K_x||2), as well as MAC key for the verification tag K_tag=H(K_x||3). The validation tag of the client is T_c=MAC(K_tag,"PACE_ECC_AES128_SHA256"||Y_c) and for the server T_s=MAC(K_tag,"PACE_ECC_AES128_SHA256"||Y_s).

4.4. Finished

The Finished messages contain the verification tags to proof that both sides derived the same cryptographic keys. In particular, the tags T_c and T_s are exchanged. After verification of the received tags, both parties are ready to establish a secure channel for application data.

5. IANA Considerations

This document request IANA to assign a new value for TLS extension type from the TLS ExtensionType Registry defined in [RFC5246] with the name "PAKE" and the number TBD1.

In addition, this document defines a new registry to be maintained by IANA:

TLS PAKEType: The registry is initially populated with the values described in Section 4.1. Future values in the range 0-63 (decimal) inclusive are assigned via Standards Action [RFC2434]. Values in the range 64-223 (decimal) inclusive are assigned via Specification Required [RFC2434]. Values from 224-255 (decimal) inclusive are reserved for Private Use [RFC2434].

6. Security Considerations

Overall, the PAKE schemes defined here preserves the structure of the TLS Handshake protocol and hence the security arguments of TLS after key-agreement carry over. The security of PAKE is discussed below.

6.1. Security and Limitations of PAKE Schemes

This memo does not make any assumptions about the strength of the shared credential, i.e. the password; best practices like a sufficient password length and use of multiple character classes SHOULD be followed. Nevertheless, an adversary can perform a dictionary attack enumerating all possible passwords. In order to prevent such attacks, implementations MUST include protection mechanisms like restricting the allowed password attempts or even locking out identities after a specific number of failed tries. Please note that this holds for server and client implementations in equal measure.

For using PAKE schemes, storing the password in plain is not required. Both parties SHOULD use a hashed version of it. In case of an augmented PAKE, the server key SHOULD be derived from the hashed password.

6.2. Security of the discussed PACE scheme

In [BFK2009], Bender et al. show the security of the PACE scheme in the model of Bellare et al. [BPR2000]. Their proof makes use of ideal ciphers and random oracles, i.e., idealized hash functions. In practice, the security of this scheme relies on the quality of the used randomness. In particular, if the nonce is not chosen in an uniform way, an adversary might use this information to compromise the password. Hence, using a strong pseudo random function to the raw random material in order to ensure the statistical quality of the random data is RECOMMENDED.

7. References

7.1. Normative References

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

[RFC2434] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", RFC 2434, DOI 10.17487/RFC2434, October 1998, <http://www.rfc-editor.org/info/rfc2434>.

[RFC3454] Hoffman, P. and M. Blanchet, "Preparation of Internationalized Strings ("stringprep")", RFC 3454, DOI 10.17487/RFC3454, December 2002, <http://www.rfc-editor.org/info/rfc3454>.

[RFC4013] Zeilenga, K., "SASLprep: Stringprep Profile for User Names and Passwords", RFC 4013, DOI 10.17487/RFC4013, February 2005, <http://www.rfc-editor.org/info/rfc4013>.

[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, DOI 10.17487/RFC4492, May 2006, <http://www.rfc-editor.org/info/rfc4492>.

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

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

7.2. Informative References

[BFK2009] Bender, J., Fischlin, M., and D. Kuegler, "Security Analysis of the PACE Key-Agreement Protocol", ISC 2009, LNCS 5735, 2009.

[BHKL2013] Bernstein, D., Hamburg, M., Krasnova, A., and T. Lange, "Elligator: Elliptic-curve points indistinguishable from uniform random strings", ACM CCS 2013, 2013.

[BPR2000] Bellare, M., Pointcheval, D., and P. Rogaway, "Authenticated Key Exchange Secure against Dictionary Attacks", Euorcrypt 2000, LNCS 1807, 2000.

[SW2006] Shallue, A. and C. Woestijne, "Construction of Rational Point on Elliptic Curves over Finite Fields", ANTS 2006, LNCS 4076, 2006.

[U2007] Ulas, M., "Rational Points on certain Hyperelliptic Curves over Finite Fields", arXiv 0706.1448, 2007.

Authors' Addresses

Dennis Kuegler
BSI
Postfach 200363
53133 Bonn
Germany

Email: Dennis.Kuegler@bsi.bund.de

Joern-Marc Schmidt
secunet Security Networks
Mergenthaler Allee 77
65760 Eschborn
Germany

Email: joern-marc.schmidt@secunet.com