

Internet Engineering Task Force  
Internet-Draft  
Intended status: Standards Track  
Expires: September 9, 2012

D. Harkins, Ed.  
Aruba Networks  
D. Halasz, Ed.  
Halasz Ventures  
March 8, 2012

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

Status of this Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

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."

The list of current Internet-Drafts can be accessed at  
<http://www.ietf.org/ietf/lid-abstracts.txt>.

The list of Internet-Draft Shadow Directories can be accessed at  
<http://www.ietf.org/shadow.html>.

This Internet-Draft will expire on September 9, 2012.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

Abstract

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

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

## Table of Contents

1. Background . . . . .	3
1.1. The Case for Certificate-less Authentication . . . . .	3
1.2. Resistance to Dictionary Attack . . . . .	3
2. Keyword Definitions . . . . .	4
3. Introduction . . . . .	4
3.1. Notation . . . . .	4
3.2. Discrete Logarithm Cryptography . . . . .	5
3.2.1. Elliptic Curve Cryptography . . . . .	5
3.2.2. Finite Field Cryptography . . . . .	6
3.3. Instantiating the Random Function . . . . .	7
3.4. Passwords . . . . .	7
3.5. Assumptions . . . . .	8
4. Specification of the TLS-PWD Handshake . . . . .	8
4.1. Fixing the Password Element . . . . .	9
4.1.1. Computing an ECC Password Element . . . . .	10
4.1.2. Computing an FFC Password Element . . . . .	11
4.2. Changes to Handshake Message Contents . . . . .	12
4.2.1. Client Hello Changes . . . . .	12
4.2.2. Server Key Exchange Changes . . . . .	13
4.2.2.1. Generation of ServerKeyExchange . . . . .	14
4.2.2.2. Processing of ServerKeyExchange . . . . .	15
4.2.3. Client Key Exchange Changes . . . . .	15
4.2.3.1. Generation of Client Key Exchange . . . . .	16
4.2.3.2. Processing of Client Key Exchange . . . . .	16
4.3. Computing the Premaster Secret . . . . .	16
5. Ciphersuite Definition . . . . .	17
6. Acknowledgements . . . . .	18
7. IANA Considerations . . . . .	18
8. Security Considerations . . . . .	19
9. Implementation Considerations . . . . .	22
10. References . . . . .	22
10.1. Normative References . . . . .	22
10.2. Informative References . . . . .	23
Authors' Addresses . . . . .	24

## 1. Background

### 1.1. The Case for Certificate-less Authentication

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

- o Frequently, TLS [RFC5246] is used in devices owned, operated, and provisioned by people who lack competency to properly use certificates and merely want to establish a secure connection using a more natural credential like a simple password. The proliferation of deployments that use a self-signed server certificate in TLS [RFC5246] followed by a PAP-style exchange over the unauthenticated channel underscores this case.
- o A password is a more natural credential than a certificate (from early childhood people learn the semantics of a shared secret), so a password-based TLS ciphersuite can be used to protect an HTTP-based certificate enrollment scheme-- e.g. an [RFC5967] -style request and an [RFC5751] -style response-- to parlay a simple password into a certificate for subsequent use with any certificate-based authentication protocol. This addresses a significant "chicken-and-egg" dilemma found with certificate-only use of [RFC5246].
- o Some PIN-code readers will transfer the entered PIN to a smart card in clear text. Assuming a hostile environment, this is a bad practice. A password-based TLS ciphersuite can enable the establishment of an authenticated connection between reader and card based on the PIN.

### 1.2. Resistance to Dictionary Attack

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

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

incorrect.

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

## 2. Keyword Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

## 3. Introduction

### 3.1. Notation

The following notation is used in this memo:

password

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

$y = H(x)$

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

$a \mid b$

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

$[a]b$

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

$x \bmod y$

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

LSB(x)

returns the least-significant bit of the bitstring "x".

### 3.2. Discrete Logarithm Cryptography

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

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

#### 3.2.1. Elliptic Curve Cryptography

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

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

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

This memo uses the following ECC Functions:

- o  $Z = \text{elem-op}(X,Y) = X + Y$ : 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.

- o  $Z = \text{scalar-op}(x,Y) = x * Y$ : 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,  $Z$ .
- o  $Y = \text{inverse}(X)$ : 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,  $R + \text{inverse}(R) = "0"$ .
- o  $z = F(X)$ : the  $x$ -coordinate of a point  $(x, y)$  on the curve is returned. This is a mapping function to convert a group element into an integer.

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

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

### 3.2.2. Finite Field Cryptography

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

- o A prime,  $p$ , determining a prime field  $GF(p)$ , the integers modulo  $p$ . The FFC group will be a subgroup of  $GF(p)^*$ , the multiplicative group of non-zero elements in  $GF(p)$ .
- o An element,  $G$ , in  $GF(p)^*$  which serves as a generator for the FFC group.  $G$  is chosen such that its multiplicative order is a sufficiently large prime divisor of  $((p-1)/2)$ .
- o A prime,  $q$ , which is the multiplicative order of  $G$ , and thus also the size of the cryptographic subgroup of  $GF(p)^*$  that is generated by  $G$ .

This memo uses the following FFC Functions:

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

- o  $Z = \text{scalar-op}(x, Y) = Y^x \bmod p$ : an integer scalar,  $x$ , acts on an FFC group element,  $Y$ , via exponentiation modulo the prime,  $p$ , to produce another FFC element,  $Z$ .
- o  $Y = \text{inverse}(X)$ : 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,  $(X * \text{inverse}(X)) \bmod p = 1$ .
- o  $z = F(X)$ : is the identity function since an element in an FFC group is already an integer. It is included here for consistency in the specification.

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

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

### 3.3. Instantiating the Random Function

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

The random function,  $H$ , in this memo is instantiated by using the hash algorithm defined by the particular TLS-PWD ciphersuite in HMAC mode with a key whose length is equal to block size of the hash algorithm and whose value is zero. For example, if the ciphersuite is TLS\_ECCPWD\_WITH\_AES\_128\_GCM\_SHA256 then  $H$  will be instantiated with SHA256 as:

$$H(x) = \text{HMAC-SHA256}([0]_{32}, x)$$

### 3.4. Passwords

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

The salting function is defined as:

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

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

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

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

### 3.5. Assumptions

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

1. The random function,  $H$ , is a "random oracle" as defined in [RANDOR].
2. The discrete logarithm problem for the chosen group is hard. That is, given  $g$ ,  $p$ , and  $y = g^x \bmod p$ , it is computationally infeasible to determine  $x$ . Similarly, for an ECC group given the curve definition, a generator  $G$ , and  $Y = x * G$ , it is computationally infeasible to determine  $x$ .
3. Quality random numbers with sufficient entropy can be created. This may entail the use of specialized hardware. If such hardware is unavailable a cryptographic mixing function (like a strong hash function) to distill entropy from multiple, uncorrelated sources of information and events may be needed. A very good discussion of this can be found in [RFC4086].

### 4. Specification of the TLS-PWD Handshake

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

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



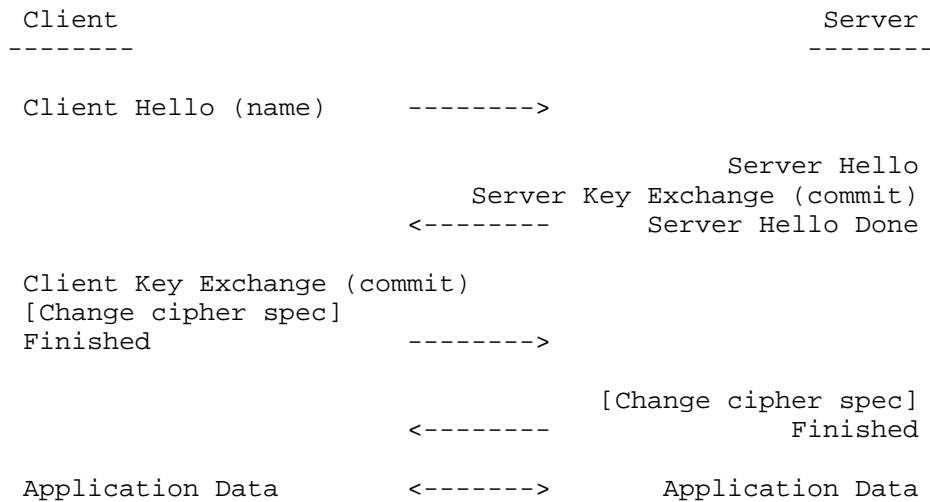


Figure 1

#### 4.1. Fixing the Password Element

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

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

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

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

$$\text{pwd-seed} = H(\text{base} \parallel \text{counter} \parallel p)$$

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

pwd-value:

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

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

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

#### 4.1.1. Computing an ECC Password Element

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

Algorithmically, the process looks like this:

```

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

```

Figure 2: Fixing PE for ECC Groups

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

#### 4.1.2. Computing an FFC Password Element

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

Algorithmically, the process looks like this:

```

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

```

Figure 3: Fixing PE for FFC Groups

## 4.2. Changes to Handshake Message Contents

### 4.2.1. Client Hello Changes

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

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

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

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

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

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

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

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

#### 4.2.2. Server Key Exchange Changes

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

Two new values have been added to the enumerated KeyExchangeAlgorithm to indicate TLS-PWD using finite field cryptography, ff\_pwd, and TLS-PWD using elliptic curve cryptography, ec\_pwd.

```
enum { ff_pwd, ec_pwd } KeyExchangeAlgorithms;

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

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

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

#### 4.2.2.1. Generation of ServerKeyExchange

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

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

$$\text{scalar} = (\text{private} + \text{mask}) \bmod q$$

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

$$\text{Element} = \text{inverse}(\text{scalar-op}(\text{mask}, \text{PE}))$$

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

##### 4.2.2.1.1. ECC Server Key Exchange

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

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

##### 4.2.2.1.2. FFC Server Key Exchange

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

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

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

#### 4.2.2.2. Processing of ServerKeyExchange

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

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

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

#### 4.2.3. Client Key Exchange Changes

When the value of KeyExchangeAlgorithm is either ff\_pwd or ec\_pwd, the ClientKeyExchange is used to convey the client's "commitment" to the server. It, therefore, contains a scalar and an Element.

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

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

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

#### 4.2.3.1. Generation of Client Key Exchange

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

For an FFC group, the scalar SHALL become the ff\_cscalar component and the Element SHALL become the ff\_celement in the FFC-specific ClientKeyExchange.

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

#### 4.2.3.2. Processing of Client Key Exchange

The server MUST validate the client's "commitment" by ensuring that: 1) the client's scalar value is greater than zero (0) and less than the order of the group,  $q$ ; and 2) that the Element is valid for the chosen group (see Section 3.2.2 and Section 3.2.1 for how to 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.

#### 4.3. Computing the Premaster Secret

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



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

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

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

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

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

## 5. Ciphersuite Definition

This memo adds the following ciphersuites:

```
CipherSuite TLS_FFCPWD_WITH_3DES_EDE_CBC_SHA = ( TBD, TBD );
CipherSuite TLS_FFCPWD_WITH_AES_128_CBC_SHA = (TBD, TBD );
CipherSuite TLS_ECCPWD_WITH_AES_128_CBC_SHA = (TBD, TBD );
CipherSuite TLS_ECCPWD_WITH_AES_128_GCM_SHA256 = (TBD, TBD );
CipherSuite TLS_ECCPWD_WITH_AES_256_GCM_SHA384 = (TBD, TBD );
CipherSuite TLS_FFCPWD_WITH_AES_128_CCM_SHA = (TBD, TBD );
CipherSuite TLS_ECCPWD_WITH_AES_128_CCM_SHA = (TBD, TBD );
CipherSuite TLS_ECCPWD_WITH_AES_128_CCM_SHA256 = (TBD, TBD );
CipherSuite TLS_ECCPWD_WITH_AES_256_CCM_SHA384 = (TBD, TBD );
```

Implementations conforming to this specification MUST support the TLS\_ECCPWD\_WITH\_AES\_128\_CCM\_SHA ciphersuite; they SHOULD support TLS\_FFCPWD\_WITH\_AES\_128\_CCM\_SHA, TLS\_FFCPWD\_WITH\_AES\_128\_CBC\_SHA,

TLS\_ECCPWD\_WITH\_AES\_128\_CBC\_SHA, TLS\_ECCPWD\_WITH\_AES\_128\_GCM\_SHA256, TLS\_ECCPWD\_WITH\_AES\_256\_GCM\_SHA384; and MAY support the remaining ciphersuites.

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

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

## 6. Acknowledgements

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

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

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

## 7. IANA Considerations

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

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

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

```
CipherSuite TLS_FFCPWD_WITH_3DES_EDE_CBC_SHA = ( TBD, TBD );  
  
CipherSuite TLS_FFCPWD_WITH_AES_128_CBC_SHA = (TBD, TBD );  
  
CipherSuite TLS_ECCPWD_WITH_AES_128_CBC_SHA = (TBD, TBD );  
  
CipherSuite TLS_ECCPWD_WITH_AES_128_GCM_SHA256 = (TBD, TBD );  
  
CipherSuite TLS_ECCPWD_WITH_AES_256_GCM_SHA384 = (TBD, TBD );  
  
CipherSuite TLS_FFCPWD_WITH_AES_128_CCM_SHA = (TBD, TBD );  
  
CipherSuite TLS_ECCPWD_WITH_AES_128_CCM_SHA = (TBD, TBD );  
  
CipherSuite TLS_ECCPWD_WITH_AES_128_CCM_SHA256 = (TBD, TBD );  
  
CipherSuite TLS_ECCPWD_WITH_AES_256_CCM_SHA384 = (TBD, TBD );
```

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

## 8. Security Considerations

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

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

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

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

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

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

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

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

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

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

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

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

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

## 9. Implementation Considerations

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

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

For example, the elliptic curve named secp256r1 (whose IANA-assigned number is 23) provides an estimated 128 bits of strength and would be compatible with an encryption algorithm supporting a key of that length, and a hash algorithm that has at least a 256-bit blocksize. Therefore, a suitable ciphersuite to use with secp256r1 could be TLS\_ECCPWD\_WITH\_AES\_128\_GCM\_SHA256.

Resistance to dictionary attack means that the attacker must launch an active attack to make a single guess at the password. If the size of the pool from which the password was extracted was  $D$ , and each password in the pool has an equal probability of being chosen, then the probability of success after a single guess is  $1/D$ . After  $X$  guesses, and removal of failed guesses from the pool of possible passwords, the probability becomes  $1/(D-X)$ . As  $X$  grows so does the probability of success. Therefore it is possible for an attacker to determine the password through repeated brute-force, active, guessing attacks. Implementations SHOULD take note of this fact and choose an appropriate pool of potential passwords-- i.e. make  $D$  big. Implementations SHOULD also take countermeasures, for instance refusing authentication attempts by a particular username for a certain amount of time, after the number of failed authentication attempts reaches a certain threshold. No such threshold or amount of time is recommended in this memo.

## 10. References

### 10.1. Normative References

- [RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

- [RFC3454] Hoffman, P. and M. Blanchet, "Preparation of Internationalized Strings ("stringprep")", RFC 3454, December 2002.
- [RFC4013] Zeilenga, K., "SASLprep: Stringprep Profile for User Names and Passwords", RFC 4013, February 2005.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [SP800-38A] National Institute of Standards and Technology, "Recommendation for Block Cipher Modes of Operation--Methods and Techniques", NIST Special Publication 800-38A, December 2001.

## 10.2. Informative References

- [RANDOR] Bellare, M. and P. Rogaway, "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols", Proceedings of the 1st ACM Conference on Computer and Communication Security, ACM Press, 1993.
- [RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.
- [RFC4492] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)", RFC 4492, May 2006.
- [RFC5751] Ramsdell, B. and S. Turner, "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification", RFC 5751, January 2010.
- [RFC5967] Turner, S., "The application/pkcs10 Media Type", RFC 5967, August 2010.
- [RFC6090] McGrew, D., Igoe, K., and M. Salter, "Fundamental Elliptic Curve Cryptography Algorithms", RFC 6090, February 2011.
- [SP800-56A] Barker, E., Johnson, D., and M. Smid, "Recommendations for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography", NIST Special Publication 800-56A, March 2007.

Authors' Addresses

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

Email: [dharkins@arubanetworks.com](mailto:dharkins@arubanetworks.com)

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

Email: [david.e.halasz@gmail.com](mailto:david.e.halasz@gmail.com)





TLS  
Internet-Draft  
Intended status: Standards Track  
Expires: January 16, 2013

S. Santesson  
3xA Security AB  
H. Tschofenig  
Nokia Siemens Networks  
July 15, 2012

Transport Layer Security (TLS) Cached Information Extension  
draft-ietf-tls-cached-info-12.txt

Abstract

Transport Layer Security (TLS) handshakes often include fairly static information, such as the server certificate and a list of trusted Certification Authorities (CAs). This information can be of considerable size, particularly if the server certificate is bundled with a complete certificate path (including all intermediary certificates up to the trust anchor public key).

This document defines an extension that omits the exchange of already available information. The TLS client informs a server of cached information, for example from a previous TLS handshake, allowing the server to omit the already available information.

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 January 16, 2013.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents

(<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

1. Introduction . . . . .	3
2. Terminology . . . . .	4
3. Cached Information Extension . . . . .	5
4. Exchange Specification . . . . .	7
4.1. Fingerprint of the Certificate Chain . . . . .	7
4.2. Fingerprint for Trusted CAs . . . . .	8
5. Example . . . . .	10
6. Security Considerations . . . . .	12
7. IANA Considerations . . . . .	13
7.1. New Entry to the TLS ExtensionType Registry . . . . .	13
7.2. New Registry for CachedInformationType . . . . .	13
8. Acknowledgments . . . . .	14
9. References . . . . .	15
9.1. Normative References . . . . .	15
9.2. Informative References . . . . .	15
Authors' Addresses . . . . .	16

## 1. Introduction

Transport Layer Security (TLS) handshakes often include fairly static information, such as the server certificate and a list of trusted Certification Authorities (CAs). This information can be of considerable size, particularly if the server certificate is bundled with a complete certificate path (including all intermediary certificates up to the trust anchor public key).

Optimizing the exchange of information to a minimum helps to improve performance in environments where devices are connected to a network with characteristics like low bandwidth, high latency and high loss rate. These types of networks exist, for example, when smart objects are connected using a low power IEEE 802.15.4 radio. For more information about the challenges with smart object deployments please see [RFC6574].

This specification defines a TLS extension that allows a client and a server to exclude transmission of cached information from the TLS handshake.

A typical example exchange may therefore look as follows. First, the TLS exchange executes the usual TLS handshake. It may decide to store the certificate provided by the server for a future exchange. When the TLS client then connects to the TLS server some time in the future, without using session resumption, it then attaches the `cached_information` extension defined in this document to the client hello message to indicate that it had cached the certificate, and it provides the fingerprint of it. If the server's certificate had not changed then the TLS server does not need to send the full certificate to the client again. In case the information had changed, the certificate payload is transmitted to the client to allow the client to update it's state information.

## 2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "MUST", "MUST NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

### 3. Cached Information Extension

This document defines a new extension type (`cached_information(TBD)`), which is used in client hello and server hello messages. The extension type is specified as follows.

```
enum {  
    cached_information(TBD), (65535)  
} ExtensionType;
```

The `extension_data` field of this extension, when included in the client hello, MUST contain the `CachedInformation` structure.

```
enum {  
    certificate_chain(1), trusted_cas(2) (255)  
} CachedInformationType;  
  
struct {  
    CachedInformationType type;  
    HashAlgorithm hash;  
    opaque hash_value<1..255>;  
} CachedObject;  
  
struct {  
    CachedObject cached_info<1..2^16-1>;  
} CachedInformation;
```

When the `CachedInformationType` identifies a `certificate_chain`, then the `hash_value` field MUST include the hash calculated over the `certificate_list` element of the Certificate payload provided by the TLS server in an earlier exchange, excluding the three length bytes of the `certificate_list` vector.

When the `CachedInformationType` identifies a `trusted_cas`, then the `hash_value` MUST include a hash calculated over the `certificate_authorities` element of the CertificateRequest payload provided by the TLS server in an earlier exchange, excluding the two length bytes of the `certificate_authorities` vector.

The hash algorithm used to calculate hash values is conveyed in the 'hash' field of the `CachedObject` element. The list of registered hash algorithms can be found in the TLS HashAlgorithm Registry, which was created by RFC 5246 [RFC5246]. The value zero (0) for 'none' is not an allowed choice for a hash algorithm and MUST NOT be used.

This document establishes a registry for `CachedInformationType` types and additional values can be added following the policy described in Section 7.

#### 4. Exchange Specification

Clients supporting this extension MAY include the "cached\_information" extension in the (extended) client hello, which MAY contain zero or more CachedObject attributes.

Server supporting this extension MAY include the "cached\_information" extension in the (extended) server hello, which MAY contain one or more CachedObject attributes. By returning the "cached\_information" extension the server indicates that it supports caching of each present CachedObject that matches the specified hash value. The server MAY support other cached objects that are not present in the extension.

Note: Clients may need the ability to cache different values depending on other information in the Client Hello that modify what values the server uses, in particular the Server Name Indication [RFC6066] value.

Following a successful exchange of "cached\_information" extensions, the server MAY send fingerprints of the cached information in the handshake exchange as a replacement for the exchange of the full data. Section 4.1 and Section 4.2 defines the syntax of the fingerprinted information.

The handshake protocol MUST proceed using the information as if it was provided in the handshake protocol. The Finished message MUST be calculated over the actual data exchanged in the handshake protocol. That is, the Finished message will be calculated over the hash values of cached information objects and not over the cached information that were omitted from transmission.

The server MUST NOT include more than one fingerprint for a single information element, i.e., at maximum only one CachedObject structure per replaced information is provided.

##### 4.1. Fingerprint of the Certificate Chain

When an object of type 'certificate\_chain' is provided in the client hello, the server MAY send a fingerprint instead of the complete certificate chain as shown below.

The original handshake message syntax is defined in RFC 5246 [RFC5246] and has the following structure:



```
opaque ASN.1Cert<1..2^24-1>;

struct {
    ASN.1Cert certificate_list<0..2^24-1>;
} Certificate;
```

By using the extension defined in this document the following information is sent:

```
struct {
    CachedObject ASN.1Cert<1..2^24-1>;
} Certificate;
```

The opaque ASN.1Cert structure is replaced with the CachedObject structure defined in this document.

Note: [I-D.ietf-tls-oob-pubkey] allows a PKIX certificate containing only the SubjectPublicKeyInfo instead of the full information typically found in a certificate. Hence, when this specification is used in combination with [I-D.ietf-tls-oob-pubkey] and the negotiated certificate type is a raw public key then the TLS server sends the hashed Certificate payload that contains a ASN.1Cert structure of the SubjectPublicKeyInfo.

#### 4.2. Fingerprint for Trusted CAs

When a hash for an object of type 'trusted\_cas' is provided in the client hello, the server MAY send a fingerprint instead of the complete certificate authorities information as shown below.

The original handshake message syntax is defined in RFC 5246 [RFC5246] and has the following structure:

```
opaque DistinguishedName<1..2^16-1>;

struct {
    ClientCertificateType certificate_types<1..2^8-1>;
    SignatureAndHashAlgorithm
        supported_signature_algorithms<2^16-1>;
    DistinguishedName certificate_authorities<0..2^16-1>;
} CertificateRequest;
```

By using the extension defined in this document the following information is sent:

```
struct {  
    ClientCertificateType certificate_types<1..2^8-1>;  
    SignatureAndHashAlgorithm  
        supported_signature_algorithms<2^16-1>;  
    CachedObject DistinguishedName<1..2^16-1>;  
} CertificateRequest;
```

The opaque DistinguishedName structure is replaced with the CachedObject structure defined in this document.

## 5. Example

Figure 1 illustrates an example exchange using the TLS cached info extension. In the normal TLS handshake exchange shown in flow (A) the TLS server provides its certificate in the Certificate payload to the client, see step [1]. This allows the client to store the certificate for future use. After some time the TLS client again interacts with the same TLS server and makes use of the TLS cached info extension, as shown in flow (B). The TLS client indicates support for this specification via the `cached_information` extension, see [2], and indicates that it has stored the `certificate_chain` from the earlier exchange. With [3] the TLS server indicates that it also supports this specification and informs the client that it also supports caching of other objects beyond the `'certificate_chain'`, namely `'trusted_cas'` (also defined in this document), and the `'foo-bar'` extension (i.e., an imaginary extension that yet needs to be defined). With [4] the TLS server provides the fingerprint of the certificate chain as described in Section 4.1.

## (A) Initial (full) Exchange

```
client_hello ->
               <-  server_hello,
                   certificate, // [1]
                   server_key_exchange,
                   server_hello_done

client_key_exchange,
change_cipher_spec,
finished
               ->
               <-  change_cipher_spec,
                   finished

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

## (B) TLS Cached Extension Usage

```
client_hello,
cached_information=(certificate_chain) -> // [2]
               <-  server_hello,
                   cached_information= // [3]
                       (certificate_chain, trusted_cas, foo-bar)
                   certificate, // [4]
                   server_key_exchange,
                   server_hello_done

client_key_exchange,
change_cipher_spec,
finished
               ->
               <-  change_cipher_spec,
                   finished

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

Figure 1: Example Message Exchange

## 6. Security Considerations

This specification defines a mechanism to reference stored state using a fingerprint. The hash algorithm used in this specification is required to have reasonable random properties in order to provide reasonably unique identifiers. There is no requirement that this hash algorithm must have strong collision resistance.

Caching information in an encrypted handshake (such as a renegotiated handshake) and sending a hash of that cached information in an unencrypted handshake might introduce integrity or data disclosure issues as it enables an attacker to identify if a known object (such as a known server certificate) has been used in previous encrypted handshakes. Information object types defined in this specification, such as server certificates, are public objects and usually not sensitive in this regard, but implementers should be aware if any cached information are subject to such security concerns and in such case SHOULD NOT send a hash over encrypted data in unencrypted handshake.

## 7. IANA Considerations

### 7.1. New Entry to the TLS ExtensionType Registry

IANA is requested to add an entry to the existing TLS ExtensionType registry, defined in RFC 5246 [RFC5246], for `cached_information(TBD)` defined in this document.

### 7.2. New Registry for CachedInformationType

IANA is requested to establish a registry for TLS CachedInformationType values. The first entries in the registry are

- o `certificate_chain(1)`
- o `trusted_cas(2)`

The policy for adding new values to this registry, following the terminology defined in RFC 5226 [RFC5226], is as follows:

- o 0-63 (decimal): Standards Action
- o 64-223 (decimal): Specification Required
- o 224-255 (decimal): reserved for Private Use

## 8. Acknowledgments

We would like to thank the following persons for your detailed document reviews:

- o Paul Wouters and Nikos Mavrogiannopoulos (December 2011)
- o Rob Stradling (February 2012)
- o Ondrej Mikle in March 2012)

Additionally, we would like to thank the TLS working group chairs, Eric Rescorla and Joe Salowey, as well as the security area directors, Sean Turner and Stephen Farrell, for their feedback and support.

## 9. References

### 9.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC3874] Housley, R., "A 224-bit One-way Hash Function: SHA-224", RFC 3874, September 2004.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [RFC6066] Eastlake, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, January 2011.

### 9.2. Informative References

- [I-D.ietf-tls-oob-pubkey]  
Wouters, P., Gilmore, J., Weiler, S., Kivinen, T., and H. Tschofenig, "TLS Out-of-Band Public Key Validation", draft-ietf-tls-oob-pubkey-03 (work in progress), April 2012.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.
- [RFC6574] Tschofenig, H. and J. Arkko, "Report from the Smart Object Workshop", RFC 6574, April 2012.



Authors' Addresses

Stefan Santesson  
3xA Security AB  
Scheelev. 17  
Lund 223 70  
Sweden

Email: sts@aaa-sec.com

Hannes Tschofenig  
Nokia Siemens Networks  
Linnoitustie 6  
Espoo 02600  
Finland

Phone: +358 (50) 4871445  
Email: Hannes.Tschofenig@gmx.net  
URI: <http://www.tschofenig.priv.at>



Network Working Group  
Internet-Draft  
Intended status: Standards Track  
Expires: January 9, 2013

Y. Pettersen  
Opera Software ASA  
July 8, 2012

The TLS Multiple Certificate Status Request Extension  
draft-ietf-tls-multiple-cert-status-extension-01

Abstract

This document defines the Transport Layer Security (TLS) Certificate Status Version 2 Extension to allow clients to specify and support multiple certificate status methods. Also defined is a new method that a server can use to provide status information (i.e., based on the Online Certificate Status Protocol and Server-Based Certificate Validation Protocol) not just about the server's own certificate, but also the status of intermediate certificates in the chain.

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 January 9, 2013.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

## 1. Introduction

The Transport Layer Security (TLS) Extension [RFC6066] framework defines, among other extensions, the Certificate Status Extension that clients can use to request the server's copy of the current status of its certificate. The benefits of this extension include a reduced number of roundtrips and network delays for the client to verify the status of the server's certificate and a reduced load on the certificate issuer's status response servers, thus solving a problem that can become significant when the issued certificate is presented by a frequently visited server.

There are two problems with the existing Certificate Status extension. First, it does not provide functionality to request the status information about intermediate Certification Authority (CA) certificates, which means the client has to request status information through other methods, such as Certificate Revocation Lists (CRLs), thus adding additional delay. Second, the current format of the extension and requirements in the TLS protocol prevents a client from offering the server multiple status methods; there are two methods available, the Online Certificate Status Protocol (OCSP) [RFC2560] and the Server-Based Certificate Validation Protocol (SCVP) [RFC5055].

Many CAs now issue intermediate CA certificates that not only specify the publication point for their CRLs in CRL Distribution Point [RFC5280], but also specify a URL for their OCSP [RFC2560] server in Authority Information Access [RFC5280]. Given that client-cached CRLs are frequently out of date, clients would benefit from using OCSP, or other protocols, to access up-to-date status information about intermediate CA certificates. The benefit to the issuing CA is less clear, as providing the bandwidth for the OCSP responder can be

costly, especially for CAs with many high-traffic subscriber sites, and this cost is a concern for many CAs. There are cases where OCSF requests for a single high-traffic site caused significant network problems for the issuing CA.

Clients will benefit from the TLS server providing certificate status information regardless of type, not just for the server certificate, but also for the intermediate CA certificates. Combining the status checks into one extension will reduce the roundtrips needed to complete the handshake by the client to just those needed for negotiating the TLS connection. Also, for the Certification Authorities, the load on their servers will depend on the number of certificates they have issued, not on the number of visitors to those sites.

For such a new system to be introduced seamlessly, clients need to be able to indicate support for the existing OCSF Certificate Status method and a new multiple-OCSF mode or the new SCVP mode.

Unfortunately, the definition of the Certificate Status extension only allows a single Certificate Status extension to be defined in a single extension record in the handshake, and the TLS Protocol [RFC5246] only allows a single record in the extension list for any given extension. This means that it is not possible for clients to indicate support for new methods while still supporting older methods, which would cause problems for interoperability between newer clients and older servers. This will not just be an issue for the multiple status request mode proposed above, but also for any other future status methods that might be introduced. This will be true not just for the current PKIX infrastructure [RFC5280], but also for alternative PKI structures.

The solution to this problem is to define a new extension, `status_request_v2`, with an extended format that allows the client to indicate support for multiple status request methods. This is implemented using a list of `CertificateStatusRequestItem` records in the extension record. As the server will select the single status method based on the selected cipher suite and the certificate presented, no significant changes are needed in the server's extension format.

### 1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

## 2. Multiple Certificate Status Extension

### 2.1. New extension

The extension defined by this document is indicated by the "status\_request\_v2" in the ExtensionType enum, which uses the following value:

```
enum {  
    status_request_v2(XX), (65535)  
} ExtensionType;
```

[[ EDITOR: The value used for status\_request\_v2 has been left as "XX". This value will be assigned when this draft progresses to RFC.]]

### 2.2. Multiple Certificate Status Request record

Clients that support a certificate status protocol (i.e., OCSP and SCVP) may send the status\_request\_v2 extension to the server in order to use the TLS handshake to transfer such data instead of downloading it through separate connections. When using this extension, the "extension\_data" field of the extension SHALL contain a CertificateStatusRequestList where:

```
struct {
    CertificateStatusType status_type;
    uint16 request_length; /* Length of request field in bytes */
    select (status_type) {
        case ocsp: OCSPStatusRequest;
        case ocsp_multi: OCSPStatusRequest;
        case scvp: SCVPStatusRequest;
    } request;
} CertificateStatusRequestItem;

enum {
    ocsp(1), ocsp_multi(YY), scvp (AA),(255)
} CertificateStatusType;

struct {
    ResponderID responder_id_list<0..2^16-1>;
    Extensions request_extensions;
} OCSPStatusRequest;

struct {
    ResponderID responder_id_list<0..2^16-1>;
    Extensions request_extensions;
} SCVPStatusRequest;

opaque ResponderID<1..2^16-1>;
opaque Extensions<0..2^16-1>;

struct {
    CertificateStatusRequestItem certificate_status_req_list<1..2^16-1>
} CertificateStatusRequestList
```

[[ EDITOR: The values used for ocsp\_multi and scvp have been left as "YY" and "AA", respectively. These values will be assigned when this draft progresses to RFC.]]

In the OCSPStatusRequest and SCVPStatusRequest structures, the "ResponderIDs" provide a list of OCSP and SCVP responders (respectively) that the client trusts. A zero-length "responder\_id\_list" sequence has the special meaning that the responders are implicitly known to the server, e.g., by prior arrangement, or are identified by the certificates used by the server. "Extensions" is a DER encoding [CCITT.X690.2002] of the OCSP and SCVP request extensions (respectively).

Both "ResponderID" and "Extensions" are DER-encoded ASN.1 types as defined in [RFC2560] (for OCSP) and [RFC5055] (for SCVP). "Extensions" is imported from [RFC5280]. A zero-length

"request\_extensions" value means that there are no extensions (as opposed to a zero-length ASN.1 SEQUENCE, which is not valid for the "Extensions" type).

In the case of the "id-pkix-ocsp-nonce" OCSP extension, [RFC2560] is unclear about its encoding; for clarification, the nonce MUST be a DER-encoded OCTET STRING, which is encapsulated as another OCTET STRING (note that implementations based on an existing OCSP client will need to be checked for conformance to this requirement).

The list of CertificateStatusRequestItem entries MUST be in order of preference.

A server that receive a client hello containing the "status\_request\_v2" extension MAY return a suitable certificate status response message to the client along with the server's certificate message. If OCSP is requested, it SHOULD use the information contained in the extension when selecting an OCSP responder and SHOULD include request\_extensions in the OCSP request.

The server returns a certificate status response along with its certificate by sending a "CertificateStatus" message immediately after the "Certificate" message (and before any "ServerKeyExchange" or "CertificateRequest" messages). If a server returns a "CertificateStatus" message in response to a status\_request\_v2 request, then the server MUST have included an extension of type "status\_request\_v2" with empty "extension\_data" in the extended server hello. The "CertificateStatus" message is conveyed using the handshake message type "certificate\_status" as follows (see also [RFC6066]):

```
struct {
    CertificateStatusType status_type;
    select (status_type) {
        case ocsp: OCSPResponse;
        case ocsp_multi: OCSPResponseList;
        case scvp: SCVPResponse;
    } response;
} CertificateStatus;

opaque OCSPResponse<0..2^24-1>;

opaque SCVPResponse<0..2^24-1>;

struct {
    OCSPResponse ocsp_response_list<1..2^24-1>
} OCSPResponseList
```



An "OCSPResponse" element contains a complete, DER-encoded OCSP response (using the ASN.1 syntax [CCITT.X680.2002] of type OCSPResponse as defined in [RFC2560]). Only one OCSP response, with a length of at least one byte, may be sent for status\_type "ocsp".

An "SCVPResponse" element contains a complete, DER-encoded SCVP response (using the ASN.1 syntax [CCITT.X680.2002] of type CVResponse as defined in [RFC5055]). Only one SCVP response, with a length of at least one byte, may be sent for status\_type "scvp". An SCVP response can include the status of intermediate certificates.

An "ocsp\_response\_list" contains a list of "OCSPResponse" elements, as specified above, each containing the OCSP response for the matching corresponding certificate in the server's Certificate TLS handshake message. That is, the first entry is the OCSP response for the first certificate in the Certificate list, the second entry is the response for the second certificate, and so on. The list MAY contain fewer OCSP responses than there were certificates in the Certificate handshake message, but there MUST NOT be more responses than there were certificates in the list. Individual elements of the list MAY have a length of 0 (zero) bytes, if the server does not have the OCSP response for that particular certificate stored, in which case, the client MUST act as if a response was not received for that particular certificate. If the client receives a "ocsp\_response\_list" that does not contain a response for one or more of the certificates in the completed certificate chain, the client SHOULD attempt to validate the certificate using an alternative retrieval method, such as downloading the relevant CRL; OCSP SHOULD in this situation only be used for the end entity certificate, not intermediate CA certificates, for reasons stated above.

Note that a server MAY also choose not to send a "CertificateStatus" message, even if it has received a "status\_request\_v2" extension in the client hello message and has sent a "status\_request\_v2" extension in the server hello message. Additionally, note that that a server MUST NOT send the "CertificateStatus" message unless it received either a "status\_request" or "status\_request\_v2" extension in the client hello message and sent a corresponding "status\_request" or "status\_request\_v2" extension in the server hello message.

Clients requesting a certificate response and receiving either one or more OCSP responses or a SCVP response in a "CertificateStatus" message MUST check the response(s) and abort the handshake, if the response is a revoked status or is otherwise not satisfactory with a bad\_certificate\_status\_response(113) alert. This alert is always fatal.

[[Open issue: At least one reviewer has suggested that the client

should treat an unsatisfactory (non-revoked) response as an empty response for that particular response and fall back to the alternative method described above]]

### 3. IANA Considerations

Section 2.1 defines the new TLS Extension `status_request_v2` enum, which should be added to the ExtensionType Values list in the IANA TLS category after IETF Consensus has decided to add the value.

Section 2.2 describes a TLS CertificateStatusType Registry to be maintained by the IANA. CertificateStatusType values are to be assigned via IETF Review as defined in [RFC5226]. The initial registry corresponds to the definition of "ExtensionType" in Section 2.2.

### 4. Security Considerations

General Security Considerations for TLS Extensions are covered in [RFC5246]. Security Considerations for the particular extension specified in this document are given below. In general, implementers should continue to monitor the state of the art and address any weaknesses identified.

#### 4.1. Security Considerations for `status_request_v2`

If a client requests an OCSP or SCVP response, it must take into account that an attacker's server using a compromised key could (and probably would) pretend not to support the extension. In this case, a client that requires OCSP or SCVP validation of certificates SHOULD either contact the OCSP or SCVP server directly or abort the handshake.

Use of the OCSP (`id-pkix-ocsp-nonce`) or SCVP nonce request extension may improve security against attacks that attempt to replay OCSP or SCVP responses; see Section 4.4.1 of [RFC2560] and Section 9 of [RFC5055] for further details.

The security considerations of [RFC2560] apply to OCSP requests and responses, and the security considerations of [RFC5055] apply to SCVP requests and responses.

### 5. Acknowledgements

This document is based on [RFC6066] authored by Donald Eastlake 3rd.

The SCVP status type description is based on text provided by Sean Turner.

## 6. Normative References

- [CCITT.X680.2002]  
International International Telephone and Telegraph Consultative Committee, "Abstract Syntax Notation One (ASN.1): Specification of basic notation", CCITT Recommendation X.680, July 2002.
- [CCITT.X690.2002]  
International International Telephone and Telegraph Consultative Committee, "ASN.1 encoding rules: Specification of basic encoding Rules (BER), Canonical encoding rules (CER) and Distinguished encoding rules (DER)", CCITT Recommendation X.690, July 2002.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC2560] Myers, M., Ankney, R., Malpani, A., Galperin, S., and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC 2560, June 1999.
- [RFC5055] Freeman, T., Housley, R., Malpani, A., Cooper, D., and W. Polk, "Server-Based Certificate Validation Protocol (SCVP)", RFC 5055, December 2007.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.
- [RFC6066] Eastlake, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, January 2011.

Author's Address

Yngve N. Pettersen  
Opera Software ASA

Email: [yngve@opera.com](mailto:yngve@opera.com)



TLS  
Internet-Draft  
Intended status: Standards Track  
Expires: January 17, 2013

P. Wouters  
Red Hat  
J. Gilmore

S. Weiler  
SPARTA, Inc.  
T. Kivinen  
AuthenTec  
H. Tschofenig  
Nokia Siemens Networks  
July 16, 2012

Out-of-Band Public Key Validation for Transport Layer Security  
draft-ietf-tls-oob-pubkey-04.txt

Abstract

This document specifies a new certificate type for exchanging raw public keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) for use with out-of-band public key validation. Currently, TLS authentication can only occur via X.509-based Public Key Infrastructure (PKI) or OpenPGP certificates. By specifying a minimum resource for raw public key exchange, implementations can use alternative public key validation methods.

One such alternative public key validation method is offered by the DNS-Based Authentication of Named Entities (DANE) together with DNS Security. Another alternative is to utilize pre-configured keys, as is the case with sensors and other embedded devices. The usage of raw public keys, instead of X.509-based certificates, leads to a smaller code footprint.

This document introduces the support for raw public keys in TLS.

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 January 17, 2013.

#### Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

1. Introduction . . . . .	4
2. Terminology . . . . .	4
3. New TLS Extensions . . . . .	4
4. TLS Handshake Extension . . . . .	5
4.1. Client Hello . . . . .	5
4.2. Server Hello . . . . .	6
4.3. Certificate Request . . . . .	6
4.4. Certificate Payload . . . . .	6
4.5. Other TLS Messages . . . . .	6
5. Examples . . . . .	6
6. Security Considerations . . . . .	9
7. IANA Considerations . . . . .	10
8. Acknowledgements . . . . .	10
9. References . . . . .	11
9.1. Normative References . . . . .	11
9.2. Informative References . . . . .	11
Authors' Addresses . . . . .	12



## 1. Introduction

Traditionally, TLS server public keys are obtained in PKIX containers in-band using the TLS handshake and validated using trust anchors based on a [PKIX] certification authority (CA). This method can add a complicated trust relationship that is difficult to validate. Examples of such complexity can be seen in [Defeating-SSL].

Alternative methods are available that allow a TLS client to obtain the TLS server public key:

- o The TLS server public key is obtained from a DNSSEC secured resource records using DANE [I-D.ietf-dane-protocol].
- o The TLS server public key is obtained from a [PKIX] certificate chain from an Lightweight Directory Access Protocol (LDAP) [LDAP] server.
- o The TLS client and server public key is provisioned into the operating system firmware image, and updated via software updates.

Some smart objects use the UDP-based Constrained Application Protocol (CoAP) [I-D.ietf-core-coap] to interact with a Web server to upload sensor data at a regular intervals, such as temperature readings. CoAP [I-D.ietf-core-coap] can utilize DTLS for securing the client-to-server communication. As part of the manufacturing process, the embedded device may be configured with the address and the public key of a dedicated CoAP server, as well as a public key for the client itself. The usage of X.509-based PKIX certificates [PKIX] does not suit all smart object deployments and would therefore be an unnecessary burden.

The Transport Layer Security (TLS) Protocol Version 1.2 [RFC5246] provides a framework for extensions to TLS as well as guidelines for designing such extensions. This document defines an extension to indicate the support for raw public keys.

## 2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

## 3. New TLS Extensions

In order to indicate the support for multiple certificate types two

new extensions are defined by this specification with the following semantic:

cert-send: The certificate payload in this message contains a certificate of the type indicated by this extension.

cert-receive: By including this extension an entity indicates that it is able to receive and process the indicated certificate types. This list is sorted by preference.

```
enum { X.509(0), RawPublicKey(1), (255) } CertType;
```

```
CertType cert-receive <1..2^8-1>;
```

```
CertType cert-send;
```

Figure 1: New TLS Extension Structures

No new cipher suites are required for use with raw public keys. All existing cipher suites that support a key exchange method compatible with the key in the certificate can be used in combination with raw public key certificate types.

#### 4. TLS Handshake Extension

This section describes the semantic of the 'cert-send' and the 'cert-receive' extensions for the different handshake messages.

##### 4.1. Client Hello

To allow a TLS client to indicate that it is able to receive a certificate of a specific type it MAY include the 'cert-receive' extension in the client hello message. To indicate the ability to process a raw public key by the server the TLS client MUST include the 'cert-receive' with the value one (1) (indicating "RawPublicKey") in the list of supported certificate types. If a TLS client only supports X.509 certificates it MAY include this extension to indicate support for it.

Future documents may define additional certificate types that require addition values to be registered.

Note: No new cipher suites are required to use raw public keys. All existing cipher suites that support a key exchange method compatible

with the defined extension can be used.

#### 4.2. Server Hello

If the server receives a client hello that contains the 'cert-receive' extension then two outcomes are possible. The server MUST either select a certificate type from client-provided list or terminate the session with a fatal alert of type "unsupported\_certificate". In the former case the procedure in Section 4.4 MUST be followed.

#### 4.3. Certificate Request

The Certificate Request payload sent by the TLS server to the TLS client MUST be accompanied by a 'cert-receive' extension, which indicates to the TLS client the certificate type the server supports.

#### 4.4. Certificate Payload

Certificate payloads MUST be accompanied by a 'cert-send' extension, which indicates the certificate format found in the Certificate payload itself.

The list of supported certificate types to choose from MUST have been obtained via the 'cert-receive' extension. This ensures that a Certificate payload only contains a certificate type that is also supported by the recipient.

When the 'RawPublicKey' certificate type is selected then the SubjectPublicKeyInfo structure MUST be placed into the Certificate payload. The type of the asymmetric key MUST match the selected key exchange algorithm.

#### 4.5. Other TLS Messages

All the other handshake messages are identical to the TLS specification.

### 5. Examples

Figure 2, Figure 3, and Figure 4 illustrate example message exchanges.

The first example shows an exchange where the TLS client indicates its ability to process two certificate types, namely raw public keys and X.509 certificates via the 'cert-receive' extension (see [1]). When the TLS server receives the client hello it processes the cert-

receive extension and since it also has a raw public key it indicates in [2] that it had chosen to place the SubjectPublicKeyInfo structure into the Certificate payload (see [3]). The client uses this raw public key in the TLS handshake and an out-of-band technique, such as DANE, to verify its validity.

```

client_hello,
cert-receive=(RawPublicKey, X.509) -> // [1]

      <-  server_hello,
          cert-send=RawPublicKey, // [2]
          certificate, // [3]
          server_key_exchange,
          server_hello_done

client_key_exchange,
change_cipher_spec,
finished                                ->

      <-  change_cipher_spec,
          finished

Application Data      <----->      Application Data

```

Figure 2: Example with Raw Public Key provided by the TLS Server

In our second example the TLS client and the TLS server use raw public keys. This is a use case envisioned for smart object networking. The TLS client in this case is an embedded device that only supports raw public keys and therefore it indicates this capability via the 'cert-receive' extension in [1]. As in the previously shown example the server fulfills the client's request and provides a raw public key into the Certificate payload back to the client (see [2] and [3]). The TLS server, however, demands client authentication and for this reason a Certificate\_Request payload is added [4], which comes with an indication of the supported certificate types by the server, see [5]. The TLS client, who has a raw public key pre-provisioned, returns it in the Certificate payload [7] to the server with the indication about its content [6].

```

client_hello,
cert-receive=(RawPublicKey) -> // [1]

                                <-  server_hello,
                                cert-send=RawPublicKey, // [2]
                                certificate, // [3]
                                certificate_request, // [4]
                                cert-receive=(RawPublicKey, X.509) // [5]
                                server_key_exchange,
                                server_hello_done

cert-send=RawPublicKey, // [6]
certificate, // [7]
client_key_exchange,
change_cipher_spec,
finished                                ->

                                <-  change_cipher_spec,
                                finished

Application Data          <----->          Application Data

```

Figure 3: Example with Raw Public Key provided by the TLS Server and the Client

In our last example we illustrate a combination of raw public key and X.509 usage. The client uses a raw public key for client authentication but the server provides an X.509 certificate. This exchange starts with the client indicating its ability to process X.509 certificates. The server provides the X.509 certificate using that format in [3] with the indication present in [2]. For client authentication, however, the server indicates in [5] that it is able to support raw public keys as well as X.509 certificates. The TLS client provides a raw public key in [7] and the indication in [6].

```

client_hello,
cert-receive=(X.509) -> // [1]

                                <-  server_hello,
                                cert-send=X.509, // [2]
                                certificate, // [3]
                                certificate_request, // [4]
                                cert-receive=(RawPublicKey, X.509) // [5]
                                server_key_exchange,
                                server_hello_done

cert-send=RawPublicKey, // [6]
certificate, // [7]
client_key_exchange,
change_cipher_spec,
finished                                ->

                                <-  change_cipher_spec,
                                finished

Application Data          <----->          Application Data

```

Figure 4: Hybrid Certificate Example

## 6. Security Considerations

The transmission of raw public keys, as described in this document, provides benefits by lowering the over-the-air transmission overhead since raw public keys are quite naturally smaller than an entire certificate. There are also advantages from a codesize point of view for parsing and processing these keys. The cryptographic procedures for associating the public key with the possession of a private key also follows standard procedures.

The main security challenge is, however, how to associate the public key with a specific entity. This information will be needed to make authorization decisions. Without a secure binding, man-in-the-middle attacks may be the consequence. This document assumes that such binding can be made out-of-band and we list a few examples in Section 1. DANE [I-D.ietf-dane-protocol] offers one such approach. If public keys are obtained using DANE, these public keys are authenticated via DNSSEC. Pre-configured keys is another out of band method for authenticating raw public keys. While pre-configured keys are not suitable for a generic Web-based e-commerce environment such keys are a reasonable approach for many smart object deployments where there is a close relationship between the software running on

the device and the server-side communication endpoint. Regardless of the chosen mechanism for out-of-band public key validation an assessment of the most suitable approach has to be made prior to the start of a deployment to ensure the security of the system.

## 7. IANA Considerations

This document defines two new TLS extension, 'cert-send' and 'cert-receive', and their values need to be added to the TLS ExtensionType registry created by RFC 5246 [RFC5246].

The values in these new extensions contains an 8-bit CertificateType field, for which a new registry, named "Certificate Types", is established in this document, to be maintained by IANA. The registry is segmented in the following way:

1. The value (0) is defined in this document.
2. Values from 2 through 223 decimal inclusive are assigned using the 'Specification Required' policy defined in RFC 5226 [RFC5226].
3. Values from 224 decimal through 255 decimal inclusive are reserved for 'Private Use', see [RFC5226].

## 8. Acknowledgements

The feedback from the TLS working group meeting at IETF#81 has substantially shaped the document and we would like to thank the meeting participants for their input. The support for hashes of public keys has been moved to [I-D.ietf-tls-cached-info] after the discussions at the IETF#82 meeting and the feedback from Eric Rescorla.

We would like to thank the following persons for their review comments: Martin Rex, Bill Frantz, Zach Shelby, Carsten Bormann, Cullen Jennings, Rene Struik, Alper Yegin, Jim Schaad, Paul Hoffman, Robert Cragie, Nikos Mavrogiannopoulos, Phil Hunt, John Bradley, and James Manger.

## 9. References

### 9.1. Normative References

- [PKIX] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

### 9.2. Informative References

- [Defeating-SSL] Marlinspike, M., "New Tricks for Defeating SSL in Practice", February 2009, <<http://www.blackhat.com/presentations/bh-dc-09/Marlinspike/BlackHat-DC-09-Marlinspike-Defeating-SSL.pdf>>.
- [I-D.ietf-core-coap] Shelby, Z., Hartke, K., Bormann, C., and B. Frank, "Constrained Application Protocol (CoAP)", draft-ietf-core-coap-10 (work in progress), June 2012.
- [I-D.ietf-dane-protocol] Hoffman, P. and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA", draft-ietf-dane-protocol-23 (work in progress), June 2012.
- [I-D.ietf-tls-cached-info] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", draft-ietf-tls-cached-info-11 (work in progress), December 2011.
- [LDAP] Sermersheim, J., "Lightweight Directory Access Protocol (LDAP): The Protocol", RFC 4511, June 2006.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.
- [RFC6091] Mavrogiannopoulos, N. and D. Gillmor, "Using OpenPGP Keys for Transport Layer Security (TLS) Authentication", RFC 6091, February 2011.



## Authors' Addresses

Paul Wouters  
Red Hat

Email: [paul@nohats.ca](mailto:paul@nohats.ca)

John Gilmore  
PO Box 170608  
San Francisco, California 94117  
USA

Phone: +1 415 221 6524  
Email: [gnu@toad.com](mailto:gnu@toad.com)  
URI: <https://www.toad.com/>

Samuel Weiler  
SPARTA, Inc.  
7110 Samuel Morse Drive  
Columbia, Maryland 21046  
US

Email: [weiler@tislabs.com](mailto:weiler@tislabs.com)

Tero Kivinen  
AuthenTec  
Eerikinkatu 28  
HELSINKI FI-00180  
FI

Email: [kivinen@iki.fi](mailto:kivinen@iki.fi)

Hannes Tschofenig  
Nokia Siemens Networks  
Linnoitustie 6  
Espoo 02600  
Finland

Phone: +358 (50) 4871445  
Email: [Hannes.Tschofenig@gmx.net](mailto:Hannes.Tschofenig@gmx.net)  
URI: <http://www.tschofenig.priv.at>



TLS  
Internet-Draft  
Intended status: Informational  
Expires: January 17, 2013

D. McGrew  
D. Wing  
Cisco  
Y. Nir  
Checkpoint  
P. Gladstone  
Independent  
July 16, 2012

TLS Proxy Server Extension  
draft-mcgrew-tls-proxy-server-01

Abstract

Transport Layer Security (TLS) is commonly used to protect HTTP and other protocols; it provides encrypted and authenticated conversations between a client and a server. In some scenarios, two TLS sessions are used, so that a third device can participate in the protected communication. In these cases, separate TLS sessions are run between the client and the middle device, on one side, and the middle device and the server on the other side. This provides the needed security, as long as the client, server, and middle device use appropriate and consistent security policies. However, this last part is problematic; how can the middle device know if a client trusts a server? At present, TLS provides no mechanism to coordinate policies, and there is no convenient way to do so.

This note defines a TLS extension that allows a TLS server to provide a TLS client with all of information about the other TLS server (or servers) that are participating in the application layer traffic that the client needs to make a well-informed access control decision. This empowers the client to reject TLS sessions that include servers that it does not trust.

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 January 17, 2013.

#### Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

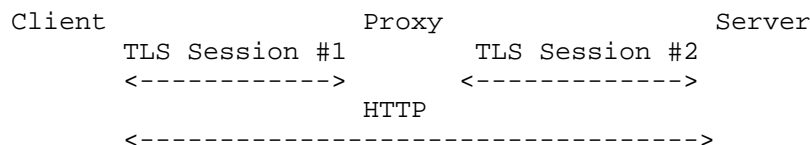
This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

1. Introduction . . . . .	4
1.1. Requirements Language . . . . .	5
2. Motivation . . . . .	5
3. Operation . . . . .	7
3.1. ProxyInfoExtension . . . . .	9
3.2. Client Certificates . . . . .	11
4. Discussion . . . . .	14
5. IANA Considerations . . . . .	14
6. Security Considerations . . . . .	15
7. References . . . . .	17
7.1. Normative References . . . . .	17
7.2. Informative References . . . . .	17
Authors' Addresses . . . . .	18

## 1. Introduction

Transport Layer Security (TLS) RFC 5246 [RFC5246] is commonly used to protect HTTP [RFC2616] as described in [RFC2818]. In some scenarios an HTTP proxy is used, for instance, to allow caching, to provide anonymity to a client, or to provide security by using an application-layer firewall to inspect the HTTP traffic on behalf of the client (e.g. to protect it against cross-site scripting attacks). A TLS session cannot protect traffic between the client and server when an HTTP proxy is present. It is possible to have separate TLS sessions between the client and the proxy, on one side, and the proxy and the server on the other side, as show in Figure 1 . This technique provides the appropriate cryptographic security (see below for a discussion of why some other alternatives are less attractive). But there is a problem: the presence of the proxy removes the client's knowledge about the server. Without this knowledge, the client has no way to decide what trust, if any, it should have in the server. This is most problematic when the client trusts multiple different servers for different applications, or trusts servers from different domains.



A proxied HTTPS session, with two independent TLS sessions.

Figure 1

A further issue is that the client cannot determine the security level of the TLS session between the proxy and the server. For instance, a client can negotiate a high security ciphersuite between itself and the proxy, but it will have no way of knowing what ciphersuite is in use between the client and the server, which could be using the obsolete 56-bit Data Encryption Standard (DES) cipher.

Another point of difficulty is the fact that there can be multiple proxies on a particular path. To solve the security issues introduced by TLS proxies in a way that is generally applicable, it is necessary to accommodate scenarios involving multiple proxies.

A separate issue is the provisioning of the proxy with information about what servers (or rather, which certificates) should be trusted. If the laptop has installed certificates that are specific to its

organization or to a particular domain, how can the proxy know to trust these certificates on behalf of the laptop?

We propose a solution in this note, by describing a TLS extension that can be used by a proxy to provide information to a TLS client about the TLS server. When this extension is used, the client is well informed about the proxy as well as the server, and can make a knowledgeable access control decision about the server, using the same processes that it uses when the proxy is not present. The data in the extension are signed by the proxy in order to bind the information about the server to a particular session between the client and the proxy. When there are multiple proxies, the client is informed about all of them. This extension also works for DTLS.

### 1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

## 2. Motivation

One motivation for the proxy extension is the benefit that it provides a client with a clear and explicit indication whenever a device is attempting to act in the role of a TLS proxy server. This allows TLS clients to reject sessions that include proxy servers that it does not trust. As an extreme example, a client could be configured to reject all sessions that involve proxy servers, in order to enforce a conservative security policy.

The following motivating example describes a typical situation with a TLS proxy, as in Figure 1. A laptop trusts the server A for a particular banking application, and trusts server B for a social media application, and can authenticate both servers by using standard PKIX certificate checking [RFC5280] and locally stored root certificates. Or rather, the client trusts a set of root certificates, and uses them to authenticate the TLS servers that it connects with. The laptop also trusts the proxy, and has a certificate by which it can authenticate the proxy. When making a connection directly with B, the laptop can authenticate the server as being trusted (that is, the server's public key appears in a certificate that has been signed by the appropriate trusted certificate authority), and it can also check the authorizations of that server (that is, B is authorized to provide the social media service, but not any other services such as banking). If the web traffic from the laptop goes through an HTTP proxy, then the proxy will need to know that it should trust both A and B to act as TLS

servers. Assuming that it does have this knowledge, it will proxy TLS connections from both A and B. However, when the client attempts to establish an HTTPS connection to A through the proxy, it has no way of knowing what security checks the proxy has applied to the connection between the proxy and A. The client cannot tell whether the trusted certificate that it associates with A was used on the connection between the proxy and A. The inability of the client to be confident of the identity of the actual server forces the client to trust all TLS servers indiscriminately.

This obstacle could be overcome by pushing the client's policy (that is, information about what servers it trusts for what applications) onto the proxy, so that the proxy can make well-informed decisions on behalf of the client. However, this alternative has significant drawbacks: it requires that the proxy obtain and store a significant amount of information about each client, and it requires the construction of a syntax by which the client's policy can be expressed and understood. In contrast, our solution moves the information about the server to the client, which does not require the communication or storage of any security policy between the client and server.

TLS proxies without this extension also defeat some recent security mechanisms that other groups have added to TLS:

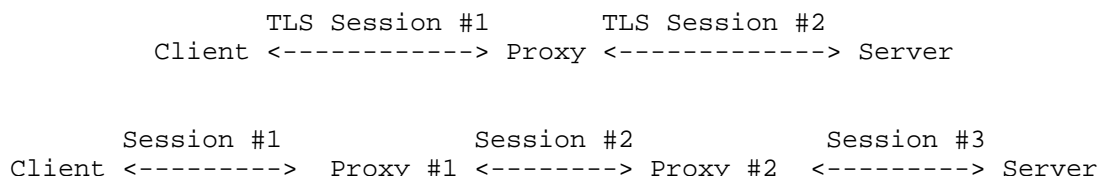
- o Extended Validation certificates ([CAB.EV]) are certificates that contain a special indication of the actual organization for which the certificate had been issued. Only a minority of public CAs are authorized to issue such certificates. By hiding the actual server certificate, proxies do not allow the browser to determine the EV status of the server certificate. This loses the visual indication that browsers typically show when EV certificates are present.
- o The DANE protocol ([I-D.ietf-dane-protocol]) stores hashes of server certificates in the DNS. Clients are expected to verify that the certificate that the server uses is the one specified in the DNS record. This fails if the certificate is one generated by the proxy.
- o Key Pinning ([I-D.ietf-websec-key-pinning]) is an alternative way to make sure the client is connecting to the correct server. Unlike DANE, that stores the certificate hash in the DNS, key pinning sends that hash in an HTTP header. Still, a client that moves behind a proxy will have stored the correct hash, but will get in TLS a certificate that does not match that hash, causing the connection to fail, unless that feature is disabled behind a proxy.



All these cases are solved if the client has access to the actual certificate sent by the server. This is provided by this extension.

### 3. Operation

In this note, a TLS proxy is a device that acts as a TLS server in one session and acts as a TLS client in another session, and passes all of the data from one session to the other, possibly modifying it in the process. That is, it is a non-transparent proxy, in the terms of [RFC2616].



A TLS session with a single proxy (top) and a TLS session with two proxies (bottom).

Figure 2

The essential idea is as follows. When a TLS proxy is contacted by a client, it does not respond to the client until it completes a TLS session with the server. It then sends the client an assertion about the server and the session, signed with the same private key that it uses in its role as the TLS proxy server. When the client receives this assertion, it checks the data in the assertion to determine whether or not it trusts the server. The assertion is carried in a ProxyInfoExtension, which is defined below.

This extension carries all of the information that is available to a TLS client about a TLS server; thus the client can use existing authorization checking processes. The client will need to verify the hostname and/or address, and check to see if the certificate has been revoked. The client authenticates the proxy server as usual during the TLS session. This ensures that the client trusts the proxy, and because of the signature on the assertion, it should trust the server certificate carried in the assertion. The proxy need not perform any checking on the server certificate, because this check is done by the client. Of course, by completing a TLS exchange with the server, the proxy verifies that the server holds the private key associated with that certificate.

If the client attempts session resumption, and the proxy can resume the session, then it must attempt to resume the session with the

server. If the server resumes the session, then the proxy must resume the session with the client. If the proxy cannot resume the session with the client, then it MUST begin a fresh session with the server, and not resume the session with the client. If the server does not resume, the proxy MUST not resume the session with the client.

Because there may be more than one proxy in any path, the TLS extension carries a list of assertions.

On receiving a ClientHello from the client, the proxy:

1. Checks for a ProxyInfoExtension in the ClientHello; if there is no such extension, then the following steps cannot be performed and are omitted,
2. Establishes a TLS session with the server (session #2 in Figure 1); a ProxyInfoExtension is included in that session,
3. Constructs a ProxyInfo structure by populating it with information about the server and the current session with that server; if the sever sends back a ProxyInfoExtension, then the ProxyInfo structure is included as the next\_proxy\_info,
4. Signs the ProxyInfo structure with the private key corresponding to the server certificate it uses in session #1,
5. Completes the session with the client (session #1 in Figure 1) and provides the ProxyInfoExtension in that session,

The proxy MAY

Perform revocation checking on the certificate chain of the server in session #2, and indicate that it has done this in the extension by setting performed\_revocation\_checking to "true".

Note that the entity acting in the role of the server in session #2 could be a proxy, but in the above it is referred to as a server because that is the role that it performs in that TLS session.

When TLS is used in HTTPS, the proxy MUST perform the Server Identity checks described in Section 3.1 of [RFC2818].

The normal operation of the proxy is to accept the (extended) ClientHello from the client and then send a ClientHello to the server. It is recommended that the TLS Proxy support commonly deployed TLS extensions (as defined in [RFC4366] et al). Any TLS extensions present on the original ClientHello MUST be examined and

either ignored, processed or forwarded (possibly after modification) to the TLS server as part of the new ClientHello.

The client:

1. Includes a ProxyInfoExtension in the ClientHello message,
2. Checks for ProxyInfoExtension in the ServerHello message; if there is no such extension, then the TLS processing continues as usual; otherwise,
3. Processes the ProxyInfo extension by checking the validity of the digitally-signed struct, then performing the usual server authentication and authorization checking on the server\_certificate\_list in the ProxyInfo,
4. Checks the revocation\_checking\_performed flag in the ProxyInfo; if it is "false", then the client SHOULD perform revocation checking on the server\_certificate\_list,
5. Checks the ProxyInfoFlag in the next\_proxy\_info field; if it is not\_empty, then the client returns to step 3 and performs that processing on the next\_proxy\_info.

In order to maintain backwards compatibility for existing TLS clients, the TLS proxies MUST (by default) perform certificate validation for the certificates that they receive from the server. The use of the ProxyInfoExtension in the extended ClientHello is an indication by the client to request the alternate processing defined by this note. In particular, if this extension is present in the extended ClientHello, then the TLS proxy should not use its own private key to dynamically generate a certificate.

The proxy will relay the data between the client and peer data connections. End-to-end flow control is maintained by the relay process: if the relay process is no longer able to write data to the destination of the relayed data, the relay process stops reading data from the source.

### 3.1. ProxyInfoExtension

The syntax of the ProxyInfo extension is as follows:

```

struct {
    ProtocolVersion      tls_version;
    CipherSuite          cipher_suite;
    CompresseionMethod   compression_method;
} ConnectionSecurityParameters;

enum { client_to_proxy, proxy_to_client,
        proxy_to_server, server_to_proxy } ProxyInfoFlag;

struct {
    select (ProxyInfoFlag) {
        case client_to_proxy:
            /* zero length body */
        case proxy_to_client:
            digitally-signed struct {
                ConnectionSecurityParameters connection_parameters;
                ASN.1Cert server_certificate_list<0..2^8-1>;
                Random server_random; /* server-side server random */
                Boolean revocation_checking_performed;
                ProxyInfo next_proxy_info;
            } SignedProxyInfo;
        case proxy_to_server:
            digitally-signed struct {
                ASN.1Cert proxy_certificate_list<0..2^8-1>;
                ProxyInfo next_proxy_info;
            } SignedProxyInfo2;
        case server_to_proxy:
            /* zero length body */
    }
} ProxyInfo;

struct {
    ProxyInfo proxy_info;
} ProxyInfoExtension;

```

The ProxyInfo structure is defined recursively, so that the signature of each proxy authenticates the information provided by the proxies that follow it on the path. The ProxyInfo contains the ProxyInfoFlag, which indicates whether or not the ProxyInfo is empty (in which case it contains no other fields) or not (in which case it contains a SignedProxyInfo structure). The SignedProxyInfo structure is signed with the public key that the proxy uses in its role as the TLS server (in session #1). That structure contains the connection\_parameters that describe the security of session #2, and the certificate chain of the server from session #2 in the server\_certificate\_list. If the proxy has performed revocation checking on that certificate chain, it indicates this by setting the

Boolean revocation\_checking\_performed. If the server in session #2 was actually a proxy itself, and it provides a ProxyInfo struct, then that struct is included in the next\_proxy\_info field. Otherwise, the next\_proxy\_info field contains an empty ProxyInfo.

```
enum {  
    /* ... */  
    proxy_info(TBD1), (65535)  
} ExtensionType;
```

### 3.2. Client Certificates

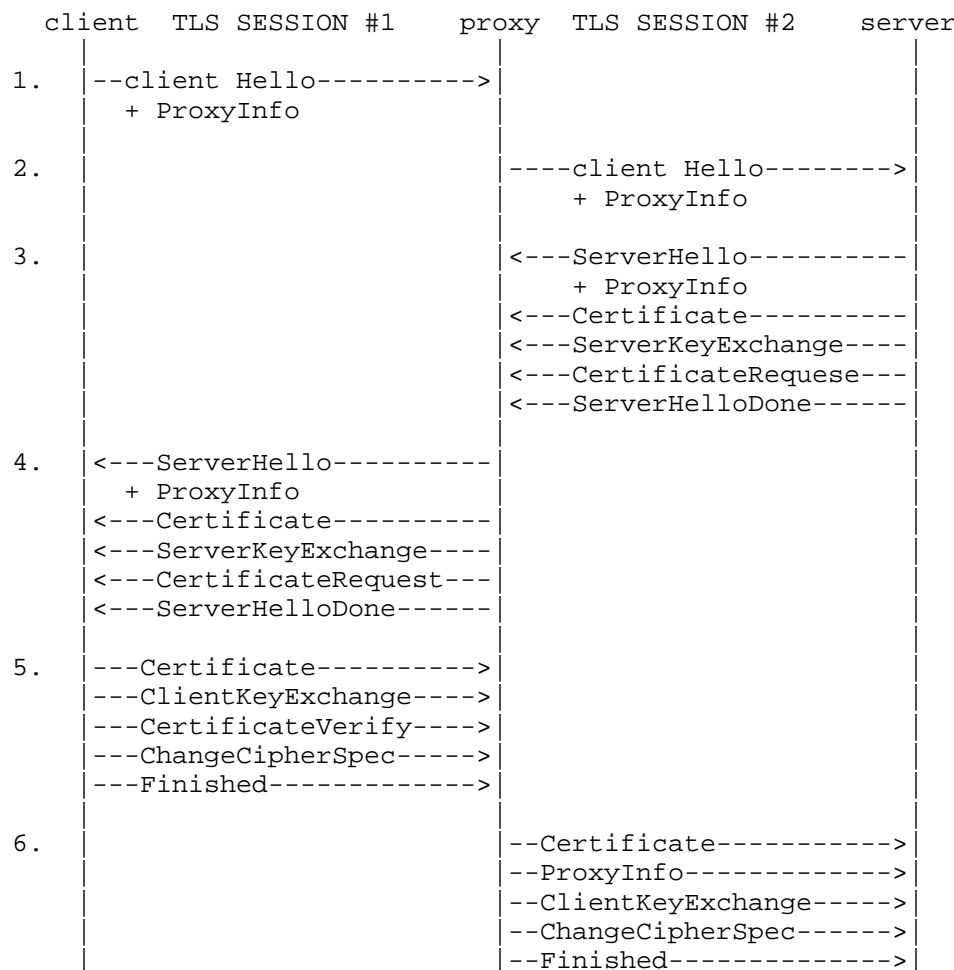
The mechanism described above supports server authentication and requires updates to the TLS client (e.g., the web browser) to support TLS proxying. Some TLS connections use client certificates (sometimes called mutual authentication). When TLS client certificates are used, the TLS server must be updated to support TLS proxy. This section describes the handshake changes beyond those described in the previous section.

On the first TLS connection to a server, the TLS proxy does not know if the TLS server will request a client certificate (that is, if the server will send a CertificateRequest in its TLS handshake). So, the TLS proxy first establishes a TCP and TLS connection to the server (as described in previous section) and when the proxy sees the TLS CertificateRequest from the server, it starts a new TCP connection. Then the following steps occur:

1. The client sends ClientHello with ProxyInfo. If the ClientHello does not include ProxyInfo, processing stops and the following steps cannot be performed.
2. The proxy, after determining the server is going to request the client's certificate (see proceeding paragraph), initiates a TLS session (TLS Session #2) to the server. The ProxyInfo in TLS Session #2 contains two ProxyInfo datastructures -- the ProxyInfo from Session #1 and the second containing the proxy's own certificate.
3. The server acknowledges it supports ProxyInfo, by including ProxyInfo in its ServerHello with a nonce it wants signed by the client, along with its CertificateRequest.
4. On Session #1, the proxy sends a ServerHello, and includes the ProxyInfo from the previous step.

5. On Session #1, the client sends its Certificate, its CertificateVerify, ChangeCipherSpec, and Finished.
6. The proxy valididates the CertificateVerify message. If it fails validation, both of the TLS sessions are abandoned.
7. On Session #2, the proxy sends the client's certificate (obtained in the previous step) in ProxyInfo, the client's CertificateVerify (obtained in the previous step), its own ClientKeyExchange, ChangeCipherSpec, and Finished.
8. The TLS server verifies the TLS client initiated the TLS communication by using the data in ProxyInfo, the nonce it sent in Step 3, and CertificateVerify.

A message flow diagram of the TLS proxy, after the proxy has determined the TLS server requests a client certificate



The ProxyCertVerify is carried from the client to the server (that is, it is un-modified by the proxy), which allows the TLS server to check that certificate. Because there are two separate TLS, Session #1 and Session #2, ProxyCertVerify cannot utilize a signature over the TLS handshake messages (as with the classic CertificateVerify). Instead, the new ProxyCertVerify message contains two signatures which provide a similar (but not identical) function. One signature is over TLS SESSION #2's ServerHello (which is conveyed into Session #1's ProxyInfo), signed using the client's private key. The second signature is over TLS SESSION #2's entire handshake, signed using the private key of the TLS proxy.

#### 4. Discussion

The ProxyInfo extension could contain information about the checking that the proxy performed on the server and its certificate. For example, if the DNS name of the server matched the subjectAltName, this fact could be indicated. It may be desirable to enumerate the ways in which the server can match its certificate, to allow the proxy to indicate to the client which of those ways was positive for a particular server.

A potential issue with the ProxyInfo extension is that it can be large, because the certificate chains that it carries can be large. Roughly speaking, the amount of certificate data presented to the client is proportional to the number of proxies on the path. It is undesirable to require that so much data be sent, but on the other hand, the client does need all of the data in order to make a well-informed access control decision. It appears that the data is the minimum required, in the sense that removing any of the data would make it impossible for the client to assess the security of the entire path.

The proxy is required to do the authentication checking on the signatures created by the server, but not the authorization checking or revocation checking. The responsibility for authorization checking is not put onto the proxy because it does not know the security policy of the client; in particular, the proxy does not know which servers the client trusts for which applications. The responsibility for revocation checking is not put onto the proxy because that process is better left to the client. The client can perform revocation checking on all of the certificate lists for all of the proxies and the server in parallel, whereas if each proxy performed the revocation checking, those processes would necessarily be serial. Since revocation checking can take a significant amount of time, the serial approach could add a significant amount of latency to the TLS session, and potentially trigger retransmissions. The parallel approach not only reduces the overall latency, but it moves it outside of the client's retransmission timer for the ClientHello message.

The ProxyInfo extension could convey the IP address of the server, or other network layer information such as the DNS name. However, it is not clear that this information is needed, so it was not included.

#### 5. IANA Considerations

This document requests IANA to update its registry of TLS extension types to assign an entry, referred herein as proxy\_info, with the



number TBD1.

## 6. Security Considerations

In a situation with a client, server, and a middle device that all need to participate in an encrypted and authenticated session, the appropriate security goals are to

- preserve the security properties of the cryptographic protocols in use,

- make the client aware of both the middle device and the server, able to authenticate the both of those devices, and able to check that both of the devices are trusted/authorized to act in their roles,

- allow the client to make access control decisions that are as well-informed as when only the client and server are present.

The idea in this note meets these goals.

We briefly describe some alternative approaches that do not meet these security goals.

First, we consider the proliferation of private keys. In order to allow one device to act as a proxy for a server, the private key of the server could be shared with the proxy. This practice may be workable when there is a one-to-one correspondence between proxies and servers, but it substantially increases the security risk. If a proxy contains multiple private keys, it becomes an attractive target for an attacker.

Second, we consider the session-key proliferation approach in which there is only a single TLS session, negotiated between the client and server, and the proxy participates in the session because either the client or the server has passed the secret session keys to the proxy (using some secure channel). One attempt at this approach is in the now abandoned [I-D.nir-tls-keyshare]. If the proxy is completely passive, and it only decrypts traffic from the TLS session and never modifies the data in that session, then this method can be secure. However, if the proxy rewrites the data inside the session, or originates messages, then the security of the TLS protocol will be undermined. Message authentication can be subverted because an attacker can intercept a message sent by the server, and forward it on to the client, bypassing the proxy. By interleaving messages sent by the proxy with ones sent by the server, an attacker can potentially confuse a client, and can certainly cause a denial of

service. Confidentiality may be undermined as well; if RC4, AES-GCM, or AES-CCM is in use, information about the plaintext will be leaked due to keystream reuse. Session-key proliferation is not secure when the proxy needs to edit the session. Most proxies do need to edit the session, and we regard it as potentially hazardous to construct a TLS proxy along these lines. Suppose that such a proxy were implemented because it was anticipated that the application proxy would be read-only, but then a future revision to the application protocol or the goals of the application proxy made it necessary to have the proxy edit the application session. If the session-key proliferation approach had been used, the implementer would be in the awkward position of having to choose between the costly path of implementing a completely new approach that preserved security, and the quick and inexpensive path of allowing the proxy to edit the session to the detriment of the security of the application.

With the ProxyInfo extension, there is no protection against the proxy lying about the security characteristics of the onward connection, unless client certificates are used. However, in any proxying scenario, it is necessary to trust the proxy, just as a client must trust the server. For instance, any proxy (not just one using the ProxyInfo extension) could choose to forward the plaintext from the session to untrusted third parties, and violate the trust of the client. It is the responsibility of the client to decide whether or not a particular device should be trusted to act in the role of proxy. The ProxyInfo proposal has the benefit of making the presence of the proxy obvious, and allows the client to refuse to deal with untrusted proxies.

Many clients use password-based authentication within a TLS tunnel. When a proxy is present, it can learn plaintext passwords, and it can gain the information needed to perform offline dictionary attacks against authentication systems that use challenge-response methods. This is a highly undesirable aspect of TLS proxying. The ProxyInfo extension does nothing to directly help this issue. However, it does indirectly improve the situation, because it empowers the client with information that enables it to reject proxies and servers that it should not trust. Since the TLS authentication (including both sever and proxy authentication) takes place before the password-based authentication, the client can protect itself by rejecting sessions with inappropriate proxies, or inappropriate servers on the path beyond the proxy.

In theory, the cryptographic proxying scenario could be considered as multiparty security negotiation and key establishment. It may be interesting to investigate such ideas because they can allow for more equitable negotiation of session parameters, and additional security properties. This note focuses on compatibility with existing

specifications and implementations, so these considerations are beyond its scope.

## 7. References

### 7.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.
- [RFC4366] Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J., and T. Wright, "Transport Layer Security (TLS) Extensions", RFC 4366, April 2006.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.

### 7.2. Informative References

- [CAB.EV] CA/Browser Forum, "GUIDELINES for the PROCESSING of EXTENDED VALIDATION CERTIFICATES", CAB GUIDELINES for the PROCESSING of EXTENDED VALIDATION CERTIFICATES, January 2009.
- [I-D.ietf-dane-protocol] Hoffman, P. and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA", draft-ietf-dane-protocol-23 (work in progress), June 2012.
- [I-D.ietf-websec-key-pinning] Evans, C. and C. Palmer, "Public Key Pinning Extension for HTTP", draft-ietf-websec-key-pinning-02 (work in progress), June 2012.
- [I-D.nir-tls-keyshare] Nir, Y., "A Method for Sharing Record Protocol Keys with a Middlebox in TLS", draft-nir-tls-keyshare-02 (work in progress), March 2012.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H.,  
Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext  
Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

Authors' Addresses

David A. McGrew  
Cisco Systems, Inc.  
510 McCarthy Blvd.  
Milpitas, CA 95035  
US

Phone: (408) 525 8651  
Email: [mcgrew@cisco.com](mailto:mcgrew@cisco.com)  
URI: <http://www.mindspring.com/~dmcgrew/dam.htm>

Dan Wing  
Cisco Systems, Inc.  
170 West Tasman Drive  
San Jose, California 95134  
USA

Email: [dwing@cisco.com](mailto:dwing@cisco.com)

Yoav  
Check Point Software Technologies Ltd.  
5 Ha'Solelim Street  
Tel Aviv, 67897  
Israel

Email: [ynir@checkpoint.com](mailto:ynir@checkpoint.com)

Philip Gladstone  
Independent



Network Working Group  
Internet-Draft  
Intended status: Informational  
Expires: January 4, 2013

Y. Pettersen  
Opera Software ASA  
July 3, 2012

Managing and removing automatic version rollback in TLS Clients  
draft-pettersen-tls-version-rollback-removal-00

Abstract

Ever since vendors started deploying TLS 1.0 clients, these clients have had to handle server implementations that do not tolerate the TLS version supported by the client, usually by automatically signaling an older supported version instead. Such version rollbacks represent a potential security hazard, if the older version should become vulnerable to attacks. The same history repeated when TLS Extensions were introduced, as some servers would not negotiate with clients that sent these protocol extensions, forcing clients to reduce protocol functionality in order to maintain interoperability.

This document outlines a procedure to help clients decide when they may use version rollback to maintain interoperability with legacy servers, under what conditions the clients should not allow version rollbacks, such as when the server has indicated support for the TLS Renegotiation Information extension. The intention of this procedure is to limit the use of automatic version rollback to legacy servers and eventually eliminate its use.

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 January 4, 2013.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## 1. Introduction

When vendors of Transport Layer Security (TLS) clients initially developed and released TLS 1.0 [RFC2246] clients, they quickly discovered that not all Secure Sockets Layer (SSL) v3 [RFC6101] servers were willing to accept or complete handshakes with the TLS clients. The reasons for this varied across various server implementations, such as not accepting versions higher than SSL v3, and various errors in the implementation of the handshake, e.g., expecting the RSA Premaster Secret's version field to match the selected version, not the signaled version.

Given the scope of the problem of getting servers fixed, in order to provide a good user experience for their customers, vendors elected instead to restart the connection and signal the older protocol version as the highest supported version in such cases.

This process was repeated when TLS Extensions [RFC6066], TLS 1.1 [RFC4346] and TLS 1.2 [RFC5246] were introduced, as clients had to disable these features to be able to connect with servers that did not tolerate them.

As a consequence, clients are not just vulnerable to a version rollback attack; in the event that a vulnerability in older protocol versions should be discovered, they are intentionally designed to be vulnerable to such attacks by automatically performing a version rollback whenever something goes wrong with the current TLS handshake.

While it would be preferable that clients do not perform version rollbacks, it is presently not practical to forbid it entirely, but there are ways to limit the use of rollbacks, and eventually phase out the usage completely.

This document presents a procedure for selecting when to allow a version rollback and how to implement it, in order to maintain interoperability with legacy servers, as well as when to not allow version rollbacks.

The main factor for deciding not to allow version rollbacks is whether the server supports the TLS Renegotiation Information Extension[RFC5746]. [RFC5746] specifically reminds implementors that servers MUST correctly handle clients that support TLS Extensions and/or new TLS versions than supported by the server. For the most part, server vendors have adhered to this, as (per July 2012) less than 0.14% of servers with Renegotiation Information extension support (70.6%) do not adhere to this requirement, compared to 4.5% among servers that does not support this extension.

### 1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

## 2. Managing Version Rollbacks

When a TLS client initially connects to a TLS server, it exchanges a number messages with the server in order to establish the encrypted connection:

- o Sending the Client Hello, which identifies the client's highest supported version, supported extensions, and cryptographic parameters
- o Receiving the Server Hello, which identifies the server's selected version, supported extensions, cryptographic parameters
- o Exchange of more messages to negotiate the encryption keys, and other parameters
- o Each sends a Finished message to the other, showing that the negotiation succeeded, after which the secure connection is active

Each step of this negotiation sequence can fail for various reasons, until the Finished messages have been sent and verified. The failures can be indicated with Alert codes or by just shutting down the connection. Frequently, many of these failures are due to incorrect implementation on either end.

This tendency toward implementation issues leading to connection



failures have caused most client vendors to adopt a policy of retrying with older versions of the protocol, in case the failure was caused by version-specific problems in the server.

### 2.1. Version Rollback Sequence

When establishing a TLS connection to a server with unknown capabilities, a client SHOULD use the following sequence, advancing to the next step if the connection attempt fails.

1. If the client supports TLS 1.1 or higher, it SHOULD send a Client Hello indicating this highest version and include all supported extensions. The version of the Record Protocol SHOULD at most be TLS 1.0
2. If step 1 failed, and the server either did not indicate a supported version or this version was TLS 1.0 or below, send a Client Hello indicating TLS 1.0 as the highest version and include all supported extensions. If this fails, the client MAY remove extensions in a separate connection attempt before considering this step to have failed.
3. If step 2 failed, and the server either did not indicate a supported version or this version was SSL v3, send a Client Hello indicating SSL v3 as the highest version, without sending TLS Extensions.

In each step, the client MUST indicate support for the TLS Renegotiation Information Extension, using the TLS\_EMPTY\_RENEGOTIATION\_INFO\_SCSV cipher suite value specified by [RFC5746] if TLS Extensions are not sent in the Client Hello.

The client MUST NOT roll back to an older version than the server has indicated, even if the connection handshake failed. That is, if the server indicates support for TLS 1.1, but the connection fails, then the client MUST NOT attempt to connect to the server using TLS 1.0, but allow the connection to fail.

### 2.2. Version Recovery

Once a connection is established and the client has received the Server Hello, it MUST check the response to determine if the server sends the TLS Renegotiation Information (RI) extension, and then decide how to proceed:

- o If the server did not return the RI extension, the client can continue the handshake as normal and MAY continue version rollbacks as described in Section 2.1 if the connection fails.

- o If the server did return the RI extension, and the client indicated its highest supported version, with extensions, (first step in Section 2.1) in the Client Hello, the client can continue the handshake as normal but MUST NOT permit version rollbacks, in case the connection fails, but instead allow the connection to fail.
- o If the server did return the RI extension, but the client was indicating an older TLS version as its highest supported version, or without TLS Extensions, the client MUST terminate the connection, reestablish it, and send a Client Hello that signals the highest supported version, and includes extensions, and it MUST NOT permit a failure to trigger a new version rollback sequence, but instead end the attempt to establish the connection.

The reason for not allowing version rollbacks if the server supports the RI extension is that such servers MUST accept that clients indicate a higher supported version than they do, and they MUST support or tolerate clients that send TLS Extensions. It must be presumed that, if such a handshake fails, it is because the connection is being subjected to a active version downgrade attack, not that the server has been incorrectly implemented in this respect.

### 3. IANA Considerations

This document makes no request of IANA.

Note to RFC Editor: this section may be removed on publication as an RFC.

### 4. Security Considerations

Allowing automatic version rollbacks exposes the TLS connection between the client and server to significant risk if the older version that gets negotiated is vulnerable to an attack that allows the transmitted information to leak.

The use of automatic version rollbacks should be limited to connections to servers that require it for interoperability reasons and be prohibited for any other servers. While it is impractical to discover which servers truly need such consideration, this document specifies the presence of the TLS Renegotiation Information extension as a proxy indication that the server does not require such interoperability considerations.

## 5. Acknowledgements

## 6. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC2246] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999.
- [RFC4346] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1", RFC 4346, April 2006.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [RFC5746] Rescorla, E., Ray, M., Dispensa, S., and N. Oskov, "Transport Layer Security (TLS) Renegotiation Indication Extension", RFC 5746, February 2010.
- [RFC6066] Eastlake, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, January 2011.
- [RFC6101] Freier, A., Karlton, P., and P. Kocher, "The Secure Sockets Layer (SSL) Protocol Version 3.0", RFC 6101, August 2011.

## Author's Address

Yngve N. Pettersen  
Opera Software ASA

Email: yngve@opera.com

