TLS
Internet-Draft
Intended status: Standards Track
Expires: August 19, 2013

P. Wouters, Ed.
Red Hat
H. Tschofenig, Ed.
Nokia Siemens Networks
J. Gilmore

S. Weiler
SPARTA, Inc.
T. Kivinen
AuthenTec
February 15, 2013

Out-of-Band Public Key Validation for Transport Layer Security (TLS) draft-ietf-tls-oob-pubkey-07.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 valiation 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.

material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 19, 2013.

Copyright Notice

Table of Contents

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:

- The TLS server public key is obtained from a DNSSEC secured resource records using DANE [RFC6698].
- The TLS server public key is obtained from a [PKIX] certificate chain from an Lightweight Directory Access Protocol (LDAP) [LDAP] server.
- 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 embeded 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] may not suit all smart object deployments and would therefore be an unneccesarry 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 registers a new value to the IANA certificate types registry for the support of 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 Extension

This section describes the changes to the TLS handshake message contents when raw public key certificates are to be used. Figure 4 illustrates the exchange of messages as described in the sub-sections below. The client and the server exchange make use of two new TLS extensions, namely 'client_certificate_type' and 'server_certificate_type', and an already available IANA TLS Certificate Type registry [TLS-Certificate-Types-Registry] to indicate their ability and desire to exchange raw public keys. These raw public keys, in the form of a SubjectPublicKeyInfo structure, are then carried inside the Certificate payload. The Certificate and the SubjectPublicKeyInfo structure is shown in Figure 1.

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

struct {
        select(certificate_type){

                // certificate type defined in this document.
        case RawPublicKey:
          opaque ASN.1_subjectPublicKeyInfo<1..2^24-1>;

                // X.509 certificate defined in RFC 5246
        case X.509:
                  ASN.1Cert certificate_list<0..2^24-1>;

                // Additional certificate type based on TLS
        // Certificate Type Registry
        };
} Certificate;
```

Figure 1: TLS Certificate Structure.

The SubjectPublicKeyInfo structure is defined in Section 4.1 of RFC 5280 [PKIX] and does not only contain the raw keys, such as the public exponent and the modulus of an RSA public key, but also an algorithm identifier. The structure, as shown in Figure 2, is encoded in an ASN.1 format and therefore contains length information as well. An example is provided in Appendix A.

```
SubjectPublicKeyInfo  ::=  SEQUENCE  {
     algorithm            AlgorithmIdentifier,
     subjectPublicKey     BIT STRING  }
```

Figure 2: SubjectPublicKeyInfo ASN.1 Structure.

The algorithm identifiers are Object Identifiers (OIDs). RFC 3279 [RFC3279], for example, defines the following OIDs shown in Figure 3.

Key Type	Document	OID
RSA	Section 2.3.1 of RFC 3279	1.2.840.113549.1.1
Digital Signature Algorithm (DSS)	Section 2.3.2 of RFC 3279	1.2.840.10040.4.1
Elliptic Curve Digital Signature Algorithm (ECDSA)	Section 2.3.5 of RFC 3279	1.2.840.10045.2.1

Figure 3: Example Algorithm Identifiers.

The message exchange in Figure 4 shows the 'client_certificate_type' and 'server_certificate_type' extensions added to the client and server hello messages.

```
    client_hello,
    client_certificate_type
    server_certificate_type   ->

                              <-  server_hello,
                                  client_certificate_type,
                                  server_certificate_type,
                                  certificate,
                                  server_key_exchange,
                                  certificate_request,
                                  server_hello_done
    certificate,
    client_key_exchange,
    certificate_verify,
    change_cipher_spec,
    finished                  ->

                              <- change_cipher_spec,
                                 finished

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

Figure 4: Basic Raw Public Key TLS Exchange.

The semantic of the two extensions is defined as follows:

The 'client_certificate_type' and 'server_certificate_type' sent in the client hello, may carry a list of supported certificate types, sorted by client preference. It is a list in the case where the client supports multiple certificate types. These extension MUST be omitted if the client only supports X.509 certificates. The 'client_certificate_type' sent in the client hello indicates the certificate types the client is able to provide to the server, when requested using a certificate_request message. The 'server_certificate_type' in the client hello indicates the type of certificates the client is able to process when provided by the server in a subsequent certificate payload.

The 'client_certificate_type' returned in the server hello indicates the certificate type found in the attached certificate payload. Only a single value is permitted. The 'server_certificate_type' in the server hello indicates the type of certificates the client is requested to provide in a subsequent certificate payload. The value conveyed in the 'server_certificate_type' MUST be selected from one of the values provided in the 'server_certificate_type' sent in the client hello. If the server does not send a certificate_request payload

or none of the certificates supported by the client (as indicated in the 'server_certificate_type' in the client hello) match the server-supported certificate types the 'server_certificate_type' payload sent in the server hello is omitted.

The "extension_data" field of this extension contains the ClientCertTypeExtension or the ServerCertTypeExtension structure, as shown in Figure 5. The CertificateType structure is an enum with with values from TLS Certificate Type Registry.

```
struct {
        select(ClientOrServerExtension)
            case client:
              CertificateType client_certificate_types<1..2^8-1>;
            case server:
              CertificateType client_certificate_type;
        }
} ClientCertTypeExtension;

struct {
        select(ClientOrServerExtension)
            case client:
              CertificateType server_certificate_types<1..2^8-1>;
            case server:
              CertificateType server_certificate_type;
        }
} ServerCertTypeExtension;
```

Figure 5: CertTypeExtension Structure.

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. TLS Handshake Extension

4.1. Client Hello

In order to indicate the support of out-of-band raw public keys, clients MUST include the 'client_certificate_type' and 'server_certificate_type' extensions extended client hello message. The hello extension mechanism is described in TLS 1.2 [RFC5246].

4.2. Server Hello

If the server receives a client hello that contains the 'client_certificate_type' and 'server_certificate_type' extensions and chooses a cipher suite then three outcomes are possible:

1. The server does not support the extension defined in this document. In this case the server returns the server hello without the extensions defined in this document.

2. The server supports the extension defined in this document and has at least one certificate type in common with the client. In this case it returns the 'server_certificate_type' and indicates the selected certificate type value.

3. The server supports the extension defined in this document but does not have a certificate type in common with the client. In this case the server terminate the session with a fatal alert of type "unsupported_certificate".

If the TLS server also requests a certificate from the client (via the certificate_request) it MUST include the 'client_certificate_type' extension with a value chosen from the list of client-supported certificates types (as provided in the 'client_certificate_type' of the client hello).

If the client indicated the support of raw public keys in the 'client_certificate_type' extension in the client hello and the server is able to provide such raw public key then the TLS server MUST place the SubjectPublicKeyInfo structure into the Certificate payload. The public key algorithm MUST match the selected key exchange algorithm.

4.3. Certificate Request

The semantics of this message remain the same as in the TLS specification.

4.4. Other Handshake Messages

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

4.5. Client authentication

Client authentication by the TLS server is supported only through authentication of the received client SubjectPublicKeyInfo via an out-of-band method.

5. Examples

Figure 6, Figure 7, and Figure 8 illustrate example exchanges.

The first example shows an exchange where the TLS client indicates its ability to receive and validate raw public keys from the server. In our example the client is quite restricted since it is unable to process other certificate types sent by the server. It also does not have credentials (at the TLS layer) it could send. The 'client_certificate_type' extension indicates this in [1]. When the TLS server receives the client hello it processes the 'client_certificate_type' extension. Since it also has a raw public key it indicates in [2] that it had choosen to place the SubjectPublicKeyInfo structure into the Certificate payload [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.

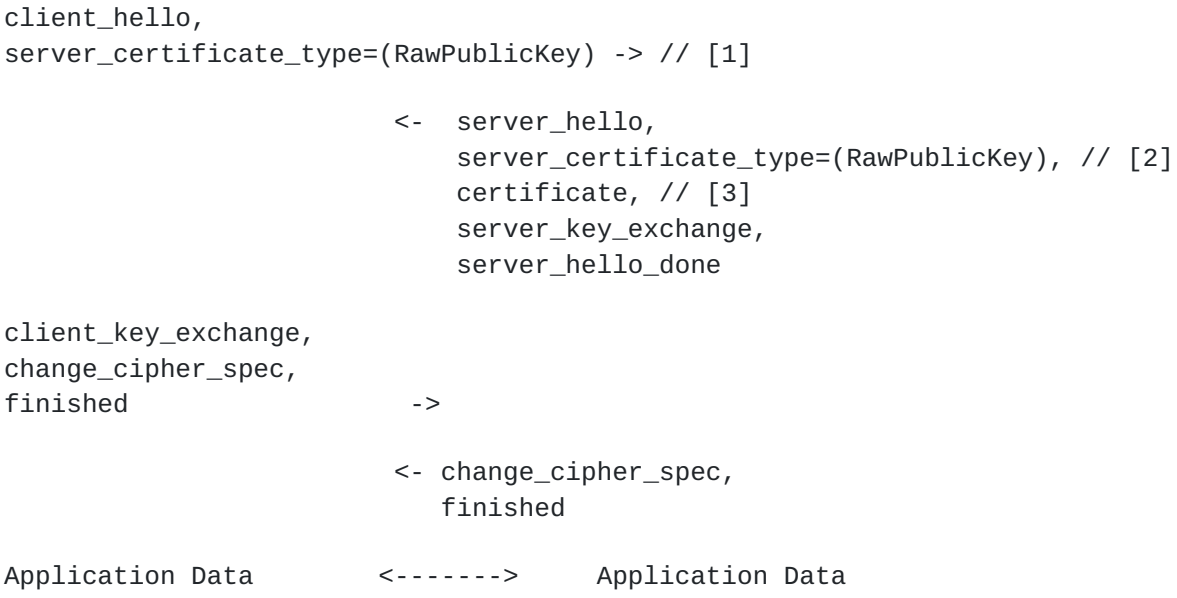

Figure 6: Example with Raw Public Key provided by the TLS Server

In our second example the TLS client as well as 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 is configured with a raw public key for use with TLS and is also able to process raw public keys sent by the server. Therefore, it indicates these capabilities in [1]. As in the previously shown example the server fulfills the client's request, indicates this via the "RawPublicKey" value in the server_certificate_type payload, and

provides a raw public key into the Certificate payload back to the client (see [3]). The TLS server, however, demands client authentication and therefore a certificate_request is added [4]. The certificate_type payload in [2] indicates that the TLS server accepts raw public keys. The TLS client, who has a raw public key pre-provisioned, returns it in the Certificate payload [5] to the server.

```
client_hello,
client_certificate_type=(RawPublicKey) // [1]
server_certificate_type=(RawPublicKey) // [1]
                         ->
                         <-  server_hello,
                             server_certificate_type=(RawPublicKey)//[2]
                             certificate, // [3]
                             client_certificate_type=(RawPublicKey)//[4]
                             certificate_request, // [4]
                             server_key_exchange,
                             server_hello_done

certificate, // [5]
client_key_exchange,
change_cipher_spec,
finished                   ->

                         <- change_cipher_spec,
                            finished

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

Figure 7: 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 provided by the server, and the ability to send raw public keys (see [1]). The server provides the X.509 certificate in [3] with the indication present in [2]. For client authentication the server indicates in [4] that it selected the raw public key format and requests a certificate from the client in [5]. The TLS client provides a raw public key in [6] after receiving and processing the TLS server hello message.

```
client_hello,
server_certificate_type=(X.509)
client_certificate_type=(RawPublicKey) // [1]
                         ->
                         <-  server_hello,
                             server_certificate_type=(X.509)//[2]
                             certificate, // [3]
                             client_certificate_type=(RawPublicKey)//[4]
                             certificate_request, // [5]
                             server_key_exchange,
                             server_hello_done
certificate, // [6]
client_key_exchange,
change_cipher_spec,
finished                  ->

                          <- change_cipher_spec,
                             finished

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

Figure 8: 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 crytographic procedures for assocating 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 [RFC6698] 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

IANA is asked to register a new value in the "TLS Certificate Types" registry of Transport Layer Security (TLS) Extensions [TLS-Certificate-Types-Registry], as follows:

Value: 2
Description: Raw Public Key
Reference: [[THIS RFC]]

This document asks IANA to allocate two new TLS extensions, "client_certificate_type" and "server_certificate_type", from the TLS ExtensionType registry defined in [RFC5246]. These extensions are used in both the client hello message and the server hello message. The new extension type is used for certificate type negotiation. The values carried in these extensions are taken from the TLS Certificate Types registry [TLS-Certificate-Types-Registry].

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.

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, Barry Leiba, Paul Hoffman, Robert Cragie, Nikos Mavrogiannopoulos, Phil Hunt, John Bradley, Klaus Hartke, Stefan Jucker, Kovatsch Matthias, Daniel Kahn Gillmor, and James Manger. Nikos Mavrogiannopoulos contributed the design for re-using the certificate type registry. Barry Leiba contributed guidance for the IANA consideration text. Stefan Jucker, Kovatsch Matthias, and Klaus Hartke provided implementation feedback regarding the SubjectPublicKeyInfo structure.

Finally, we would like to thank our TLS working group chairs, Eric Rescorla and Joe Salowey, for their guidance and support.

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.

[TLS-Certificate-Types-Registry] "TLS Certificate Types Registry", February 2013, <http://www.iana.org/assignments/tls-extensiontype-values#tls-extensiontype-values-2>.

9.2. Informative References

[ASN.1-Dump] Gutmann, P., "ASN.1 Object Dump Program", February 2013, <http://www.cs.auckland.ac.nz/~pgut001/>.

[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-13 (work in progress), December 2012.

[I-D.ietf-tls-cached-info] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", draft-ietf-tls-cached-info-13 (work in progress), September 2012.

[LDAP] Sermersheim, J., "Lightweight Directory Access Protocol (LDAP): The Protocol", RFC 4511, June 2006.

[RFC3279] Bassham, L., Polk, W., and R. Housley, "Algorithms and Identifiers for the Internet X.509 Public Key

Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 3279, April 2002.

[RFC6698] Hoffman, P. and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA", RFC 6698, August 2012.

Appendix A. Example Encoding

For example, the following hex sequence describes a SubjectPublicKeyInfo structure inside the certificate payload:

```
       0     1     2     3     4     5     6     7     8     9
---+------+-----+-----+-----+-----+-----+-----+-----+-----+-----
1  | 0x30, 0x81, 0x9f, 0x30, 0x0d, 0x06, 0x09, 0x2a, 0x86, 0x48,
2  | 0x86, 0xf7, 0x0d, 0x01, 0x01, 0x01, 0x05, 0x00, 0x03, 0x81,
3  | 0x8d, 0x00, 0x30, 0x81, 0x89, 0x02, 0x81, 0x81, 0x00, 0xcd,
4  | 0xfd, 0x89, 0x48, 0xbe, 0x36, 0xb9, 0x95, 0x76, 0xd4, 0x13,
5  | 0x30, 0x0e, 0xbf, 0xb2, 0xed, 0x67, 0x0a, 0xc0, 0x16, 0x3f,
6  | 0x51, 0x09, 0x9d, 0x29, 0x2f, 0xb2, 0x6d, 0x3f, 0x3e, 0x6c,
7  | 0x2f, 0x90, 0x80, 0xa1, 0x71, 0xdf, 0xbe, 0x38, 0xc5, 0xcb,
8  | 0xa9, 0x9a, 0x40, 0x14, 0x90, 0x0a, 0xf9, 0xb7, 0x07, 0x0b,
9  | 0xe1, 0xda, 0xe7, 0x09, 0xbf, 0x0d, 0x57, 0x41, 0x86, 0x60,
10 | 0xa1, 0xc1, 0x27, 0x91, 0x5b, 0x0a, 0x98, 0x46, 0x1b, 0xf6,
11 | 0xa2, 0x84, 0xf8, 0x65, 0xc7, 0xce, 0x2d, 0x96, 0x17, 0xaa,
12 | 0x91, 0xf8, 0x61, 0x04, 0x50, 0x70, 0xeb, 0xb4, 0x43, 0xb7,
13 | 0xdc, 0x9a, 0xcc, 0x31, 0x01, 0x14, 0xd4, 0xcd, 0xcc, 0xc2,
14 | 0x37, 0x6d, 0x69, 0x82, 0xd6, 0xc6, 0xc4, 0xbe, 0xf2, 0x34,
15 | 0xa5, 0xc9, 0xa6, 0x19, 0x53, 0x32, 0x7a, 0x86, 0x0e, 0x91,
16 | 0x82, 0x0f, 0xa1, 0x42, 0x54, 0xaa, 0x01, 0x02, 0x03, 0x01,
17 | 0x00, 0x01
```

Figure 9: Example SubjectPublicKeyInfo Structure Byte Sequence.

The decoded byte-sequence shown in Figure 9 (for example using Peter's ASN.1 decoder [ASN.1-Dump]) illustrates the structure, as shown in Figure 10.

```
   Offset  Length   Description
   ----------------------------------------------------------------
      0     3+159:   SEQUENCE {
      3      2+13:     SEQUENCE {
      5       2+9:      OBJECT IDENTIFIER Value (1 2 840 113549 1 1 1)
                :             PKCS #1, rsaEncryption
     16       2+0:      NULL
                :       }
     18     3+141:    BIT STRING, encapsulates {
     22     3+137:      SEQUENCE {
     25     3+129:        INTEGER Value (1024 bit)
    157       2+3:        INTEGER Value (65537)
                :       }
                :     }
                :   }
```

Figure 10: Decoding of Example SubjectPublicKeyInfo Structure.

Authors' Addresses

Paul Wouters (editor)
Red Hat

Email: paul@nohats.ca

Hannes Tschofenig (editor)
Nokia Siemens Networks
Linnoitustie 6
Espoo 02600
Finland

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

John Gilmore
PO Box 170608
San Francisco, California 94117
USA

Phone: +1 415 221 6524
Email: gnu@toad.com
URI: https://www.toad.com/

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

Email: weiler@tislabs.com

Tero Kivinen
AuthenTec
Eerikinkatu 28
HELSINKI FI-00180
FI

Email: kivinen@iki.fi