TLS
Internet-Draft
Intended status: Standards Track
Expires: April 22, 2014

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

S. Weiler
SPARTA, Inc.
T. Kivinen
AuthenTec

October 19, 2013

Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)

draft-ietf-tls-oob-pubkey-10.txt

Abstract

This document specifies a new certificate type and two TLS extensions 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.

Status of This Memo

This Internet-Draft will expire on April 22, 2014.

Copyright Notice

Table of Contents

1. Introduction

Traditionally, TLS client and server public keys are obtained in PKIX containers in-band as part of the TLS handshake procedure and are validated using trust anchors based on a PKIX certification authority (CA) [RFC5280]. 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 clients/servers to obtain the TLS servers/client public key:

- o TLS clients can obtain the TLS server public key from a DNSSEC secured resource records using DANE [RFC6698].

- The TLS client or server public key is obtained from a certificate chain via a Lightweight Directory Access Protocol (LDAP) [RFC4511] server or web page.

- The TLS client and server public key is provisioned into the operating system firmware image, and updated via software updates. For example:

  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 /private key pair for the client itself.

This document introduces the use of raw public keys in TLS/DTLS. Raw public key thereby means that only a sub-set of the information found in typical certificates is utilized, namely the SubjectPublicKeyInfo structure of a PKIX certificates that carries the parameters necessary to describe the public key. Other parameters also found in a PKIX certificate are omitted. A consequence of omitting various certificate related structures is that the resulting raw public key is fairly small (compared to the original certificate) and does not require codepaths for the ASN.1 parser, for certificate path validation and other PKIX related processing tasks. To further reduce the size of the exchanged information this specification can be combined with the TLS Cached Info extension [I-D.ietf-tls-cached-info], which enables TLS endpoints to just exchange fingerprints of their public keys (rather than the full public keys).

The mechanism defined herein only provides authentication when an out-of-band mechanism is also used to bind the public key to the entity presenting the key.

This document is structured as follows: Section 3 defines the structure of the two new TLS extensions "client_certificate_type" and "server_certificate_type", which can be used as part of an extended

TLS handshake when raw public keys are to be used. Section 4 defines the behavior of the TLS client and the TLS server. Example exchanges are described in Section 5. Finally, in Section 7 this document also 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].

We use the terms 'TLS server' and 'server' as well as 'TLS client' and 'client' interchangable.

3. Structure of the Raw Public Key Extension

This section defines the two TLS extensions 'client_certificate_type' and 'server_certificate_type', which can be used as part of an extended TLS handshake when raw public keys are used. Section 4 defines the behavior of the TLS client and the TLS server using this extension.

This specification reuses the SubjectPublicKeyInfo structure to encode the raw public key and to convey that information within the TLS handshake the Certificate payload is utilized as a container, as shown in Figure 1. The shown Certificate structure is an adaptation of its original form [RFC5246].

```
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: Certificate Payload as a Container for the Raw Public Key.

The SubjectPublicKeyInfo structure is defined in Section 4.1 of RFC 5280 [RFC5280] 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 algorithm identifier can also include parameters. The structure, as shown in Figure 2, is represented in a DER encoded ASN.1 format [X.690] and therefore contains length information as well. An example is provided in Appendix A.

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

       AlgorithmIdentifier   ::=  SEQUENCE  {
            algorithm               OBJECT IDENTIFIER,
            parameters              ANY DEFINED BY algorithm OPTIONAL  }
```

Figure 2: SubjectPublicKeyInfo ASN.1 Structure.

The algorithm identifiers are Object Identifiers (OIDs). RFC 3279 [RFC3279] and [RFC5480], for example, define the following OIDs shown in Figure 3. Note that this list is not exhaustive and more OIDs may be defined in future RFCs. RFC 5480 also defines a number of OIDs.

Key Type	Document	OID
RSA	Section 2.3.1 of RFC 3279	1.2.840.113549.1.1
Digital Signature		

Algorithm (DSA)	Section 2.3.2 of RFC 3279	1.2.840.10040.4.1
Elliptic Curve Digital Signature Algorithm (ECDSA)	Section 2 of RFC 5480	1.2.840.10045.2.1

Figure 3: Example Algorithm Object Identifiers.

The extension format for extended client hellos and extended server, via the "extension_data" field, is used to carry the ClientCertTypeExtension and the ServerCertTypeExtension structures. These two structures are shown in Figure 4. The CertificateType structure is an enum with values taken from the 'TLS Certificate Type' registry [TLS-Certificate-Types-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 4: CertTypeExtension Structure.

4. TLS Client and Server Handshake Behavior

This specification extends the ClientHello and the ServerHello messages, according to the extension procedures defined in [RFC5246]. It does not extend or modify any other TLS message.

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.

The high-level message exchange in Figure 5 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 5: Basic Raw Public Key TLS Exchange.

4.1. Client Hello

In order to indicate the support of raw public keys, clients include the 'client_certificate_type' and/or the 'server_certificate_type' extensions in an extended client hello message. The hello extension mechanism is described in Section 7.4.1.4 of TLS 1.2 [RFC5246].

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 types of certificates the client is able to process when provided by the server in a subsequent certificate payload.

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.

The TLS client MUST omit the 'client_certificate_type' extension in the client hello if it does not possess a client certificate or is not configured to use one with the given TLS server. The TLS client MUST omit the 'server_certificate_type' extension in the client hello if it is unable to process any certificate types from the server (which is a situation that should not occur in normal circumstances).

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 but it does not have a certificate type in common with the client. Then the server terminates the session with a fatal alert of type "unsupported_certificate".

3. The server supports the extensions defined in this document and has at least one certificate type in common with the client. In this case the processing rules described below are followed.

If the client hello indicates support of raw public keys in the 'client_certificate_type' extension and the server chooses to use raw public keys then the TLS server MUST place the SubjectPublicKeyInfo structure into the Certificate payload.

If the TLS server also requests a certificate from the client (via the certificate_request message) 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 server does not send a certificate_request payload (for example, because client authentication happens at the application layer or no client authentication is required) 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 then the 'server_certificate_type' payload in the server hello is omitted.

4.3. Client Authentication

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

5. Examples

This section illustrates a number of possible usage scenarios.

5.1. TLS Server uses Raw Public Key

This section shows an example 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 to the server and therefore omits the 'client_certificate_type' extension. Hence, the client only populates the 'server_certificate_type' extension with the raw public key type, as shown in [1].

When the TLS server receives the client hello it processes the extension. Since it has a raw public key it indicates in [2] that it had chosen to place the SubjectPublicKeyInfo structure into the Certificate payload [3].

The client uses this raw public key in the TLS handshake together with an out-of-band validation technique, such as DANE, to verify it.

```
   client_hello,
   server_certificate_type=(RawPublicKey) // [1]
                            ->
                            <-  server_hello,
                                server_certificate_type=(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 6: Example with Raw Public Key provided by the TLS Server.

5.2. TLS Client and Server use Raw Public Keys

This section shows an example where 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.

5.3. Combined Usage of Raw Public Keys and X.509 Certificate

This section shows an example combining raw public keys and X.509 certificates. 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 code size 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. Without a secure binding between identity and key, the protocol will be vulnerable to masquerade and man-in-the-middle attacks. 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. In order to address these vulnerabilities, specifications that make use of the extension MUST specify how the identity and public key are bound. In addition to ensuring the binding is done out-of-band an implementation also needs to check the status of that binding.

> 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.

A downgrading attack is another possibility for an adversary to gain advantages. Thereby, an attacker might try to influence the handshake exchange to make the parties select different certificate types than they would normally choose.

For this attack, an attacker must actively change one or more handshake messages. If this occurs, the client and server will

compute different values for the handshake message hashes. As a result, the parties will not accept each others' Finished messages. Without the master_secret, the attacker cannot repair the Finished messages, so the attack will be discovered.

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, Peter Sylvester, Hauke Mehrtens, Alexey Melnikov, 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.

Christer Holmberg provided the General Area (Gen-Art) review, Yaron Sheffer provided the Security Directorate (SecDir) review, Bert

Greevenbosch provided the Applications Area Directorate review, and Linda Dunbar provided the Operations Directorate review.

We would like to thank our TLS working group chairs, Eric Rescorla and Joe Salowey, for their guidance and support. Finally, we would like to thank Sean Turner, who is the responsible security area director for this work for his review comments and suggestions.

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.

[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.

[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.

[RFC5480] Turner, S., Brown, D., Yiu, K., Housley, R., and T. Polk, "Elliptic Curve Cryptography Subject Public Key Information", RFC 5480, March 2009.

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

[X.690] ITU, "ITU-T Recommendation X.690 (2002) | ISO/IEC 8825-1:2002, Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", 2002.

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., and C. Bormann, "Constrained Application Protocol (CoAP)", draft-ietf-core-coap-18 (work in progress), June 2013.

[I-D.ietf-tls-cached-info]
Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", draft-ietf-tls-cached-info-15 (work in progress), October 2013.

[RFC4511] Sermersheim, J., "Lightweight Directory Access Protocol (LDAP): The Protocol", RFC 4511, June 2006.

[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

The following example 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.

We used Peter Gutmann's ASN.1 decoder [ASN.1-Dump] to turn the above-shown byte-sequence into an ASN.1 structure, as shown in of the 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 Solutions and 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