Internet Engineering Task Force (IETF)
Request for Comments: 7250
Category: Standards Track
ISSN: 2070-1721

P. Wouters, Ed.
Red Hat
H. Tschofenig, Ed.
ARM Ltd.
J. Gilmore
Electronic Frontier Foundation
S. Weiler
Parsons
T. Kivinen
INSIDE Secure

June 2014

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

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). The new certificate type allows raw public keys to be used for authentication.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc7250.

Table of Contents

1. Introduction 3
2. Terminology 4
3. Structure of the Raw Public Key Extension 4
4. TLS Client and Server Handshake Behavior 7
4.1. Client Hello 7
4.2. Server Hello 8
4.3. Client Authentication 9
4.4. Server Authentication 9
5. Examples 10
5.1. TLS Server Uses a Raw Public Key 10
5.2. TLS Client and Server Use Raw Public Keys 11
5.3. Combined Usage of Raw Public Keys and X.509 Certificates 12
6. Security Considerations 13
7. IANA Considerations 14
8. Acknowledgements 14
9. References 15
9.1. Normative References 15
9.2. Informative References 15
Appendix A. Example Encoding 17

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). This method can add a complicated trust relationship that is difficult to validate. Examples of such complexity can be seen in [Defeating-SSL]. TLS is, however, also commonly used with self-signed certificates in smaller deployments where the self-signed certificates are distributed to all involved protocol endpoints out-of-band. This practice does, however, still require the overhead of the certificate generation even though none of the information found in the certificate is actually used.

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

- The TLS client can obtain the TLS server public key from a DNSSEC-secured resource record using DNS-Based Authentication of Named Entities (DANE) [RFC6698].

- The TLS client or server public key is obtained from a [PKIX] certificate chain from a Lightweight Directory Access Protocol [LDAP] 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] to interact with a Web server to upload sensor data at regular intervals, such as temperature readings. 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. With raw public keys, only a subset of the information found in typical certificates is utilized: namely, the SubjectPublicKeyInfo structure of a PKIX certificate that carries the parameters necessary to describe the public key. Other parameters found in PKIX certificates are omitted. By omitting various certificate-related structures, the resulting raw public key is kept fairly small in comparison to the original certificate, and the code to process the keys can be simpler. Only a minimalistic ASN.1 parser is needed; code for certificate path validation and other PKIX-related

processing is not required. Note, however, the SubjectPublicKeyInfo structure is still in an ASN.1 format. To further reduce the size of the exchanged information, this specification can be combined with the TLS Cached Info extension [CACHED-INFO], which enables TLS peers to exchange just fingerprints of their 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.

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. Section 6 describes security considerations with this approach. Finally, in Section 7 this document registers a new value to the IANA "TLS Certificate Types" subregistry 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" interchangeably.

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 these extensions.

This specification uses raw public keys whereby the already available encoding used in a PKIX certificate in the form of a SubjectPublicKeyInfo structure is reused. To carry the raw public key within the TLS handshake, the Certificate payload is used 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 Types" subregistry
    };
} 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 [PKIX] and not only contains 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 SubjectPublicKeyInfo value in the Certificate payload MUST contain the DER encoding [X.690] of the SubjectPublicKeyInfo. The structure, as shown in Figure 2, therefore also contains length information. 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 RFC 5480 [RFC5480], for example, define the OIDs shown in Figure 3. Note that this list is not exhaustive, and more OIDs may be defined in future RFCs.

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 and server hellos, which uses 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 Types" subregistry of the "Transport Layer Security (TLS) Extensions" registry [TLS-Ext-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 extension 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 extension 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 extensions sent in the client hello each carry a list of supported certificate types, sorted by client preference. When the client supports only one certificate type, it is a list containing a single element.

The TLS client MUST omit certificate types from the client_certificate_type extension in the client hello if it does not possess the corresponding raw public key or certificate that it can provide to the server when requested using a certificate_request message, or if it is not configured to use one with the given TLS server. If the client has no remaining certificate types to send in the client hello, other than the default X.509 type, it MUST omit the client_certificate_type extension in the client hello.

The TLS client MUST omit certificate types from the server_certificate_type extension in the client hello if it is unable to process the corresponding raw public key or other certificate type. If the client has no remaining certificate types to send in the client hello, other than the default X.509 certificate type, it MUST omit the entire server_certificate_type extension from the client hello.

4.2. Server Hello

If the server receives a client hello that contains the client_certificate_type extension and/or the server_certificate_type extension, 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 any 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.

The client_certificate_type extension in the client hello indicates the certificate types the client is able to provide to the server, when requested using a certificate_request message. If the TLS

server wants to request a certificate from the client (via the certificate_request message), it MUST include the client_certificate_type extension in the server hello. This client_certificate_type extension in the server hello then indicates the type of certificates the client is requested to provide in a subsequent certificate payload. The value conveyed in the client_certificate_type extension MUST be selected from one of the values provided in the client_certificate_type extension sent in the client hello. The server MUST also include a certificate_request payload in the server hello message.

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 client_certificate_type extension in the client hello) match the server-supported certificate types, then the client_certificate_type payload in the server hello MUST be omitted.

The server_certificate_type extension 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. If the client hello indicates support of raw public keys in the server_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. With the server_certificate_type extension in the server hello, the TLS server indicates the certificate type carried in the Certificate payload. This additional indication enables avoiding parsing ambiguities since the Certificate payload may contain either the X.509 certificate or a SubjectPublicKeyInfo structure. Note that only a single value is permitted in the server_certificate_type extension when carried in the server hello.

4.3. Client Authentication

When the TLS server has specified RawPublicKey as the client_certificate_type, 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.

4.4. Server Authentication

When the TLS server has specified RawPublicKey as the server_certificate_type, authentication of the TLS server to the TLS client is supported only through authentication of the received client SubjectPublicKeyInfo via an out-of-band method.

5. Examples

Figures 6, 7, and 8 illustrate example exchanges. Note that TLS ciphersuites using a Diffie-Hellman exchange offering forward secrecy can be used with a raw public key, although this document does not show the information exchange at that level with the subsequent message flows.

5.1. TLS Server Uses a Raw Public Key

This section shows an example where the TLS client indicates its ability to receive and validate a raw public key from the server. In this 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 one of the use cases 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 a raw public key 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 (2), and provides a raw public key in the Certificate payload back to the client (see (3)). The TLS server demands client authentication, and therefore includes a certificate_request (4). The client_certificate_type payload in (5) indicates that the TLS server accepts a raw public key. The TLS client, which has a raw public key pre-provisioned, returns it in the Certificate payload (6) 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 // (5)
                              certificate_request, // (4)
                              server_key_exchange,
                              server_hello_done

certificate, // (6)
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 Certificates

This section shows an example combining a raw public key and an X.509 certificate. The client uses a raw public key for client authentication, and the server provides an X.509 certificate. This exchange starts with the client indicating its ability to process an X.509 certificate, OpenPGP certificate, or a raw public key, if provided by the server. It prefers a raw public key, since the RawPublicKey value precedes the other values in the server_certificate_type vector. Additionally, the client indicates that it has a raw public key for client-side authentication (see (1)). The server chooses to provide its X.509 certificate in (3) and indicates that choice in (2). For client authentication, the server indicates in (4) that it has 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=(RawPublicKey, X.509, OpenPGP)
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 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.

However, the main security challenge is how to associate the public key with a specific entity. Without a secure binding between identifier and key, the protocol will be vulnerable to 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 need to specify how the identifier 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. Using 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.

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 has registered a new value in the "TLS Certificate Types" subregistry of the "Transport Layer Security (TLS) Extensions" registry [TLS-Ext-Registry], as follows:

Value: 2
Description: Raw Public Key
Reference: RFC 7250

IANA has allocated two new TLS extensions, client_certificate_type and server_certificate_type, from the "TLS ExtensionType Values" subregistry defined in [RFC5246]. These extensions are used in both the client hello message and the server hello message. The new extension types are used for certificate type negotiation. The values carried in these extensions are taken from the "TLS Certificate Types" subregistry of the "Transport Layer Security (TLS) Extensions" registry [TLS-Ext-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 [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, Stephen Farrell, Richard Barnes, and James Manger. Nikos Mavrogiannopoulos contributed the design for reusing the certificate type registry. Barry Leiba contributed guidance for the IANA Considerations 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

[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.

[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.

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

[TLS-Ext-Registry]
IANA, "Transport Layer Security (TLS) Extensions", <http://www.iana.org/assignments/tls-extensiontype-values>.

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

9.2. Informative References

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

[CACHED-INFO]
Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", Work in Progress, February 2014.

[CoAP] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, June 2014.

[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>.

[LDAP] 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

For example, the hex sequence shown in Figure 9 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 Gutmann'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: pwouters@redhat.com

Hannes Tschofenig (editor)
ARM Ltd.
6060 Hall in Tirol
Austria

EMail: Hannes.tschofenig@gmx.net
URI: http://www.tschofenig.priv.at

John Gilmore
Electronic Frontier Foundation
PO Box 170608
San Francisco, California 94117
USA

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

Samuel Weiler
Parsons
7110 Samuel Morse Drive
Columbia, Maryland 21046
US

EMail: weiler@tislabs.com

Tero Kivinen
INSIDE Secure
Eerikinkatu 28
Helsinki FI-00180
FI

EMail: kivinen@iki.fi