

TLS  
Internet-Draft  
Intended status: Standards Track  
Expires: January 31, 2014

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

S. Weiler  
SPARTA, Inc.  
T. Kivinen  
AuthenTec  
July 30, 2013

Out-of-Band Public Key Validation for Transport Layer Security (TLS)  
draft-ietf-tls-oob-pubkey-09.txt

## Abstract

This document specifies a new certificate type and two TLS extensions, one for the client and one for the server, 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 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 31, 2014.

## Copyright Notice

Copyright (c) 2013 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

<a href="#">1.</a>	Introduction . . . . .	<a href="#">2</a>
<a href="#">2.</a>	Terminology . . . . .	<a href="#">3</a>
<a href="#">3.</a>	New TLS Extension . . . . .	<a href="#">3</a>
<a href="#">4.</a>	TLS Handshake Extension . . . . .	<a href="#">7</a>
<a href="#">4.1.</a>	Client Hello . . . . .	<a href="#">7</a>
<a href="#">4.2.</a>	Server Hello . . . . .	<a href="#">7</a>
<a href="#">4.3.</a>	Certificate Request . . . . .	<a href="#">7</a>
<a href="#">4.4.</a>	Other Handshake Messages . . . . .	<a href="#">8</a>
<a href="#">4.5.</a>	Client authentication . . . . .	<a href="#">8</a>
<a href="#">5.</a>	Examples . . . . .	<a href="#">8</a>
<a href="#">6.</a>	Security Considerations . . . . .	<a href="#">10</a>
<a href="#">7.</a>	IANA Considerations . . . . .	<a href="#">11</a>
<a href="#">8.</a>	Acknowledgements . . . . .	<a href="#">12</a>
<a href="#">9.</a>	References . . . . .	<a href="#">12</a>
<a href="#">9.1.</a>	Normative References . . . . .	<a href="#">12</a>
<a href="#">9.2.</a>	Informative References . . . . .	<a href="#">13</a>
<a href="#">Appendix A.</a>	Example Encoding . . . . .	<a href="#">13</a>
	Authors' Addresses . . . . .	<a href="#">14</a>

## [1.](#) Introduction

Traditionally, TLS client and 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 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](#)].

- o The TLS client or server public key is obtained from a [\[PKIX\]](#) certificate chain from an Lightweight Directory Access Protocol (LDAP) [\[LDAP\]](#) server or web page.

- o 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 key for the client itself.

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 registers a new value to the IANA certificate types registry for the support of raw public keys. It also defines two new TLS extensions, "client\_certificate\_type" and "server\_certificate\_type".

## [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 keys 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 algorithm identifier can also include parameters. The structure, as shown in Figure 2, is encoded in an 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	<a href="#">Section 2.3.1 of RFC 3279</a>	1.2.840.113549.1.1
.....	.....	.....
Digital Signature Algorithm (DSS)	<a href="#">Section 2.3.2 of RFC 3279</a>	1.2.840.10040.4.1

.....	.....	.....
Elliptic Curve Digital Signature Algorithm (ECDSA)	<a href="#">Section 2 of RFC 5480</a>	1.2.840.10045.2.1

Figure 3: Example Algorithm Object 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,

```

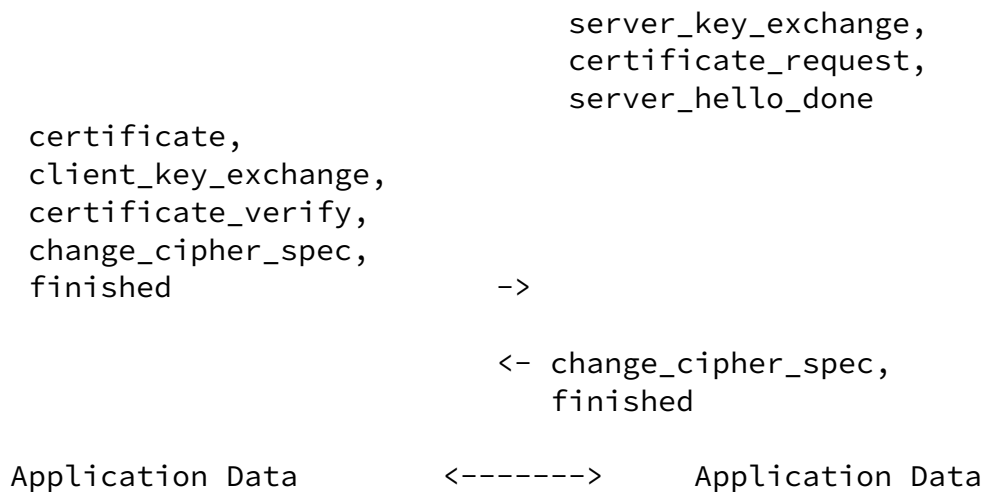


Figure 4: Basic Raw Public Key TLS Exchange.

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 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 in an 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 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.

#### [4.3.](#) Certificate Request



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 chosen 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.

```
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

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 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.

## [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, 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.

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.

Wouters, et al.

Expires January 31, 2014

[Page 12]

---

Internet-Draft

TLS 00B Public Key Validation

July 2013

- [TLS-Certificate-Types-Registry]  
, "TLS Certificate Types Registry", February 2013, <<http://www.iana.org/assignments/tls-extensiontype-values#tls-extensiontype-values-2>>.
- [X.690] , "Information technology - ASN.1 encoding rules: > Specification of Basic Encoding Rules (BER), Canonical > Encoding Rules (CER) and Distinguished Encoding Rules > (DER).", [RFC 5280](#), 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-14](#) (work in progress), March 2013.

[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 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

Wouters, et al.

Expires January 31, 2014

[Page 14]

Internet-Draft

TLS OOB Public Key Validation

July 2013

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](mailto:weiler@tislabs.com)

Tero Kivinen  
AuthenTec  
Eerikinkatu 28  
HELSINKI FI-00180  
FI

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