

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

S. Santesson
3xA Security AB
H. Tschofenig
Nokia Siemens Networks
September 12, 2012

Transport Layer Security (TLS) Cached Information Extension
draft-ietf-tls-cached-info-13.txt

Abstract

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

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

Status of this Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on March 16, 2013.

Copyright Notice

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

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

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

Table of Contents

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

1. Introduction

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

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

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

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

2. Terminology

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

3. Cached Information Extension

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

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

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

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

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

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

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

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

4. Exchange Specification

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

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

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

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

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

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

4.1. Fingerprint of the Certificate Chain

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

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

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

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

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

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

The `certificate_list` vector of opaque `ASN.1Cert` elements in the original syntax is replaced with a vector holding `CachedObject` structures as defined in this document.

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

4.2. Fingerprint for Trusted CAs

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

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

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

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

By using the extension defined in this document the following

information is sent:

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

The `certificate_authorities` vector of opaque `DistinguishedName` elements in the original syntax is replaced with a vector holding `CachedObject` structures as defined in this document.

5. Example

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

(A) Initial (full) Exchange

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

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

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

(B) TLS Cached Extension Usage

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

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

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

Figure 1: Example Message Exchange

6. Security Considerations

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

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

7. IANA Considerations

7.1. New Entry to the TLS ExtensionType Registry

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

7.2. New Registry for CachedInformationType

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

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

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

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

8. Acknowledgments

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

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

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

9. References

9.1. Normative References

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

9.2. Informative References

- [I-D.ietf-tls-oob-pubkey]
Wouters, P., Gilmore, J., Weiler, S., Kivinen, T., and H. Tschofenig, "Out-of-Band Public Key Validation for Transport Layer Security", draft-ietf-tls-oob-pubkey-04 (work in progress), July 2012.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.
- [RFC6574] Tschofenig, H. and J. Arkko, "Report from the Smart Object Workshop", RFC 6574, April 2012.

Authors' Addresses

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

Email: sts@aaa-sec.com

Hannes Tschofenig
Nokia Siemens Networks
Linnoitustie 6
Espoo 02600
Finland

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

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 22, 2013

Y. Pettersen
Opera Software ASA
October 19, 2012

The TLS Multiple Certificate Status Request Extension
draft-ietf-tls-multiple-cert-status-extension-02

Abstract

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

Status of this Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 22, 2013.

Copyright Notice

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

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

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

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

1. Introduction

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

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

Many CAs are now issuing intermediate CA certificates that not only specify the publication point for their CRLs in a CRL Distribution Point [RFC5280], but also specify a URL for their OCSP [RFC2560] server in Authority Information Access [RFC5280]. Given that client-cached CRLs are frequently out of date, clients would benefit from using OCSP to access up-to-date status information about intermediate CA certificates. The benefit to the issuing CA is less clear, as providing the bandwidth for the OCSP responder can be costly, especially for CAs with many high-traffic subscriber sites, and this cost is a concern for many CAs. There are cases where OCSP requests for a single high-traffic site caused significant network problems

for the issuing CA.

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

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

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

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

1.1. Requirements Language

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

2. Multiple Certificate Status Extension

2.1. New extension

The extension defined by this document is indicated by the "status_request_v2" in the ExtensionType enum, which uses the following value:

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

[[EDITOR: The value used for status_request_v2 has been left as "XX". This value will be assigned when this draft progresses to RFC.]]

2.2. Multiple Certificate Status Request record

Clients that support a certificate status protocol like OCSP may send the status_request_v2 extension to the server in order to use the TLS handshake to transfer such data instead of downloading it through separate connections. When using this extension, the "extension_data" field of the extension SHALL contain a CertificateStatusRequestList where:

```
struct {  
    CertificateStatusType status_type;  
    uint16 request_length; /* Length of request field in bytes */  
    select (status_type) {  
        case ocsp: OCSPStatusRequest;  
        case ocsp_multi: OCSPStatusRequest;  
    } request;  
} CertificateStatusRequestItem;  
  
enum { ocsp(1), ocsp_multi(YY), (255) } CertificateStatusType;  
  
struct {  
    ResponderID responder_id_list<0..2^16-1>;  
    Extensions request_extensions;  
} OCSPStatusRequest;  
  
opaque ResponderID<1..2^16-1>;  
opaque Extensions<0..2^16-1>;  
  
struct {  
    CertificateStatusRequestItem certificate_status_req_list<1..2^16-1>;  
} CertificateStatusRequestList;
```

[[EDITOR: The value used for ocsp_multi has been left as YY. This

value will be assigned when this draft progresses to RFC.]]

In the OCSPStatusRequest, the "ResponderIDs" provides a list of OCSP responders that the client trusts. A zero-length "responder_id_list" sequence has the special meaning that the responders are implicitly known to the server, e.g., by prior arrangement, or are identified by the certificates used by the server. "Extensions" is a DER encoding [CCITT.X690.2002] of the OCSP request extensions.

Both "ResponderID" and "Extensions" are DER-encoded ASN.1 types as defined in [RFC2560]. "Extensions" is imported from [RFC5280]. A zero-length "request_extensions" value means that there are no extensions (as opposed to a zero-length ASN.1 SEQUENCE, which is not valid for the "Extensions" type).

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

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

A server that receive a client hello containing the "status_request_v2" extension MAY return a suitable certificate status response message to the client along with the server's certificate message. If OCSP is requested, it SHOULD use the information contained in the extension when selecting an OCSP responder and SHOULD include request_extensions in the OCSP request.

The server returns a certificate status response along with its certificate by sending a "CertificateStatus" message immediately after the "Certificate" message (and before any "ServerKeyExchange" or "CertificateRequest" messages). If a server returns a "CertificateStatus" message in response to a status_request_v2 request, then the server MUST have included an extension of type "status_request_v2" with empty "extension_data" in the extended server hello. The "CertificateStatus" message is conveyed using the handshake message type "certificate_status" as follows (see also [RFC6066]):

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

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

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

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

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

Note that a server MAY also choose not to send a "CertificateStatus" message, even if it has received a "status_request_v2" extension in the client hello message and has sent a "status_request_v2" extension in the server hello message. Additionally, note that that a server MUST NOT send the "CertificateStatus" message unless it received either a "status_request" or "status_request_v2" extension in the client hello message and sent a corresponding "status_request" or "status_request_v2" extension in the server hello message.

Clients requesting an OCSP response and receiving one or more OCSP responses in a "CertificateStatus" message MUST check the OCSP response(s) and abort the handshake, if the response is a revoked status or is otherwise not satisfactory with a `bad_certificate_status_response(113)` alert. This alert is always fatal.

[[Open issue: At least one reviewer has suggested that the client should treat an unsatisfactory (non-revoked) response as an empty response for that particular response and fall back to the alternative method described above]]

3. IANA Considerations

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

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

4. Security Considerations

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

4.1. Security Considerations for `status_request_v2`

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

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

The security considerations of [RFC2560] apply to OCSP requests and responses.

5. Acknowledgements

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

6. Normative References

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

Author's Address

Yngve N. Pettersen
Opera Software ASA

Email: yngve@opera.com

TLS
Internet-Draft
Intended status: Standards Track
Expires: April 25, 2013

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

S. Weiler
SPARTA, Inc.
T. Kivinen
AuthenTec
October 22, 2012

Out-of-Band Public Key Validation for Transport Layer Security (TLS)
draft-ietf-tls-oob-pubkey-06.txt

Abstract

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

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

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

Status of this Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference

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

This Internet-Draft will expire on April 25, 2013.

Copyright Notice

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

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

Table of Contents

1. Introduction	4
2. Terminology	4
3. New TLS Extension	4
4. TLS Handshake Extension	7
4.1. Client Hello	7
4.2. Server Hello	7
4.3. Certificate Request	8
4.4. Other Handshake Messages	8
4.5. Client authentication	8
5. Examples	8
6. Security Considerations	11
7. IANA Considerations	12
8. Acknowledgements	12
9. References	13
9.1. Normative References	13
9.2. Informative References	13
Authors' Addresses	14

1. Introduction

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

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

- o The TLS server public key is obtained from a DNSSEC secured resource records using DANE [RFC6698].
- o The TLS server public key is obtained from a [PKIX] certificate chain from an Lightweight Directory Access Protocol (LDAP) [LDAP] server.
- o The TLS client and server public key is provisioned into the operating system firmware image, and updated via software updates.

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

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

2. Terminology

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

3. New TLS Extension

This section describes the changes to the TLS handshake message

contents when raw public key certificates are to be used. Figure 3 illustrates the exchange of messages as described in the sub-sections below. The client and the server exchange the newly defined `certificate_type` extension 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 `SubjectPublicKeyInfo` structure is defined in Section 4.1 of RFC 5280. Note that the `SubjectPublicKeyInfo` block 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 1, is encoded in an ASN.1 format and therefore contains length information as well.

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

Figure 1: SubjectPublicKeyInfo ASN.1 Structure.

The algorithm identifiers are Object Identifiers (OIDs). RFC 3279 [RFC3279], for example, defines the following OIDs shown in Figure 2.

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 2: Example Algorithm Identifiers.

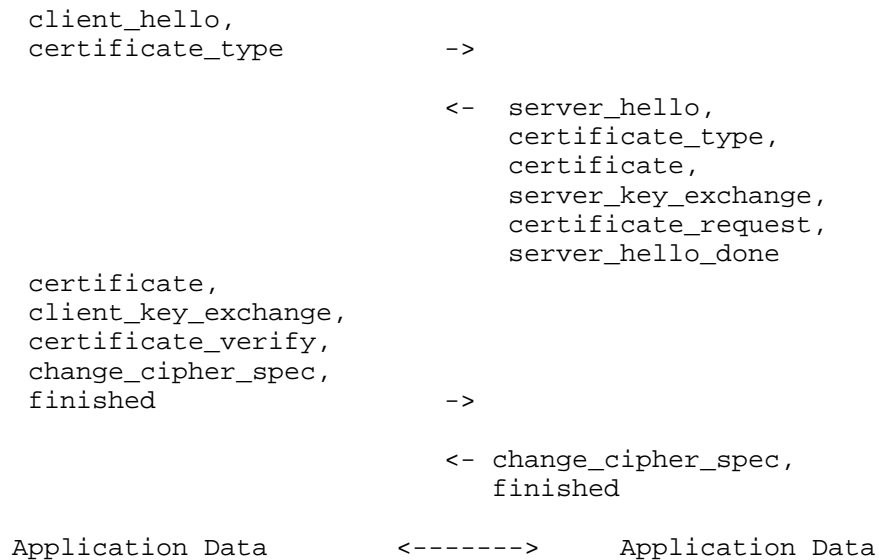


Figure 3: Basic Raw Public Key TLS Exchange.

The "certificate_type" TLS extension carries a list of supported certificate types the client can send and receive, sorted by client preference. Two values are defined for each certificate types to differentiate whether a client or a server is able to process a certificate of a specific type or can also send it. This extension MUST be omitted if the client only supports X.509 certificates. The "extension_data" field of this extension contains a CertTypeExtension structure.

Note that the CertTypeExtension structure is being used both by the client and the server, even though the structure is only specified once in this document.

The structure of the CertTypeExtension is defined as follows:

```
enum { client, server } ClientOrServerExtension;

enum { X.509-Accept (0),
       X.509-Offer (1),
       RawPublicKey-Accept (2),
       RawPublicKey-Offer (3),
       (255)
       } CertificateType;

struct {
    select(ClientOrServerExtension)
        case client:
            CertificateType certificate_types<1..2^8-1>;
        case server:
            CertificateType certificate_type;
    }
} CertTypeExtension;
```

Figure 4: CertTypeExtension Structure.

The '-Offer' postfix indicates that a TLS entity is able to send the indicated certificate type to the other communication partner. The '-Accept' postfix indicates that a TLS entity is able to receive the indicated certificate type.

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 an extension of type "certificate_type" to the extended client hello message. The "certificate_type" TLS extension is assigned the value of [TBD] from the TLS ExtensionType registry. This value is used as the extension number for the extensions in both the client hello message and the server 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 "certificate_type" extension and chooses a cipher suite then two outcomes are possible. The server MUST either select a certificate

type from the `CertificateType` field in the extended client hello or terminate the session with a fatal alert of type `"unsupported_certificate"`.

The certificate type selected by the server is encoded in a `CertTypeExtension` structure, which is included in the extended server hello message using an extension of type `"certificate_type"`. Servers that only support X.509 certificates MAY omit including the `"certificate_type"` extension in the extended server hello.

If the client supports the receipt of raw public keys and the server is able to provide such a raw public key then the TLS server MUST place the `SubjectPublicKeyInfo` structure into the `Certificate` payload. The public key 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 5, Figure 6, and Figure 7 illustrate example exchanges.

The first example shows an exchange where the TLS client indicates its ability to receive raw public keys. This client is quite restricted since it is unable to process other certificate types sent by the server. It also does not have credentials it could send. The `'certificate_type'` extension indicates this in [1]. When the TLS server receives the client hello it processes the `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,
certificate_type=(RawPublicKey-Accept) -> // [1]

      <- server_hello,
          certificate_type=(RawPublicKey-Offer), // [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 5: 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 the 'certificate_type' extension in [1]. As in the previously shown example the server fulfills the client's request, indicates this via the 'RawPublicKey-Offer' in the 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,
certificate_type=(RawPublicKey-Offer, RawPublicKey-Accept) -> // [1]

        <-  server_hello,
            certificate_type=(RawPublicKey-Offer,
                            RawPublicKey-Accept) // [2]
            certificate, // [3]
            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 6: 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. The server provides the X.509 certificate in [3] with the indication present in [2]. For client authentication, however, the server indicates in [2] that it is able to support raw public keys and requests a certificate from the client in [4]. The TLS client provides a raw public key in [5] after receiving and processing the TLS server hello message.

```

client_hello,
certificate_type=(X.509-Accept, RawPublicKey-Offer) -> // [1]

        <-  server_hello,
            certificate_type=(X.509-Offer,
                RawPublicKey-Accept), // [2]
            certificate, // [3]
            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: Hybrid Certificate Example

6. Security Considerations

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

The main security challenge is, however, how to associate the public key with a specific entity. This information will be needed to make authorization decisions. Without a secure binding, man-in-the-middle attacks may be the consequence. This document assumes that such binding can be made out-of-band and we list a few examples in Section 1. DANE [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

This document defines a new TLS extension, "certificate_type", assigned a value of [TBD] from the TLS ExtensionType registry defined in [RFC5246]. This value is used as the extension number for the extensions in both the client hello message and the server hello message. The new extension type is used for certificate type negotiation.

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

1. The values 0 - 3 are defined in Figure 4.
2. Values from 3 through 223 decimal inclusive are assigned via IETF Consensus [RFC5226].
3. Values from 224 decimal through 255 decimal inclusive are reserved for Private Use [RFC5226].

8. Acknowledgements

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

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

9. References

9.1. Normative References

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

9.2. Informative References

- [Defeating-SSL] Marlinspike, M., "New Tricks for Defeating SSL in Practice", February 2009, <<http://www.blackhat.com/presentations/bh-dc-09/Marlinspike/BlackHat-DC-09-Marlinspike-Defeating-SSL.pdf>>.
- [I-D.ietf-core-coap] Shelby, Z., Hartke, K., Bormann, C., and B. Frank, "Constrained Application Protocol (CoAP)", draft-ietf-core-coap-12 (work in progress), October 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] Serpersheim, 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.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.
- [RFC6698] Hoffman, P. and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA", RFC 6698, August 2012.

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@tislab.com

Tero Kivinen
AuthenTec
Eerikinkatu 28
HELSINKI FI-00180
FI

Email: kivinen@iki.fi

TLS Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 18, 2013

S. Keoh
O. Garcia-Morchon
S. Kumar
E. Dijk
Philips Research
October 15, 2012

DTLS-based Multicast Security for Low-Power and Lossy Networks (LLNs)
draft-keoh-tls-multicast-security-00

Abstract

Wireless IP-based systems will be increasingly used for building control systems in the future where wireless devices interconnect with each other, forming low-power and lossy networks (LLNs). The CoAP/6LoWPAN standards are emerging as the de-facto protocols in this area for resource-constrained devices. Both multicast and security are key needs in these networks. This draft presents a method for securing multicast communication in LLNs based on the DTLS security protocol which is already present in CoAP devices. This is achieved by using unicast DTLS-protected communication channel to distribute keying material and security parameters to group members. Group keys consisting of a Traffic Encryption Key (TEK) and a Traffic Authentication Key (TAK) are generated by group members based on the keying material received. A group member uses its DTLS record layer implementation to encrypt a multicast message and provide message authentication using the group keys before sending the message via IP multicast to the group.

Requirements Language

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

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 April 18, 2013.

Copyright Notice

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

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

Table of Contents

1. Introduction	4
1.1. Terminology	5
1.2. Outline	6
2. Use Cases and Requirements	6
2.1. Use Cases	6
2.2. Security Requirements	7
3. Overview of DTLS-based Secure Multicast	9
3.1. IP Multicast	9
3.2. Securing Multicast in LLNs	10
4. Multicast Group Keys Generation and Distribution	11
4.1. DTLS based Group Security Association (GSA)	11
4.2. Generation of Group Keys	13
5. Multicast Data Security	15
5.1. Sending Secure Multicast Messages	15
5.2. Receiving Secure Multicast Messages	16
6. Group Keys Renewal	16
7. IANA Considerations	16
8. Security Considerations	17
9. Acknowledgements	17
10. References	17
10.1. Normative References	17
10.2. Informative References	17
Authors' Addresses	19

1. Introduction

There is an increased use of wireless control networks in city infrastructure, environmental monitoring, industrial automation, and building management systems. This is mainly driven by the fact that the independence from physical control wires allows for freedom of placement, portability and for reducing the cost of installation as less cable placement and drilling are required. Consequently, there is an ever growing number of electronic devices, sensors and actuators that have become Internet connected, thus creating a trend towards Internet of Things (IoT). These connected devices are equipped with communication capability that enables them to interact with each other as well as with Internet services at anytime and anywhere. However, the devices in such wireless control networks are usually battery-operated or powered by scavenged energy, they have limited computational resources (low CPU clock, small RAM and flash storage) and often, the communication bandwidth is limited (e.g., IEEE 802.15.4 radio), and also the transmission is unreliable. Hence, such wireless control networks are also known as Low-power and Lossy Networks (LLNs).

In addition to the usual device-to-device unicast communication that would allow devices to interact with each other, group communication is an important feature in LLNs that can be effectively used to convey messages to a group of devices without requiring the sender to perform time- and energy-consuming multiple unicast transmissions to reach group members. For example, in a building control management system, Heating, Ventilation and Air-Conditioning (HVAC) and lighting devices can be grouped according to the layout of the building, and control commands can be issued to a group of devices. Group communication for LLNs has been made possible using the Constrained Application Protocol (CoAP) [I-D.ietf-core-coap] based on IP-multicast.

Currently, CoAP can be protected using Datagram Transport Layer Security (DTLS) [RFC4347]. However, DTLS is mainly used to secure a connection between two endpoints and it cannot be used to protect multicast group communication. We believe that group communication in LLNs is equally important and should be secured as it is also vulnerable to the usual attacks over the air (eavesdropping, tampering, message forgery, replay, etc). Although there have been a lot of efforts in IETF to standardize mechanisms to secure multicast communication, they are not necessarily suitable for LLNs which have much more limited bandwidth and resources. For example, the MIKEY Architecture [RFC3830] is mainly designed to facilitate multimedia distribution, while TESLA [RFC4082] is proposed as a protocol for broadcast authentication of the source and not for protecting the confidentiality of multicast messages.

This draft describes an approach to use DTLS as mandated in CoAP to support multicast security. The secure channel established with DTLS is used to distribute keying material (including a TEK Generation Key (TGK), security parameters, multicast security policy) to members of a multicast group, which then allows a group member to securely generate group keys, known as Traffic Encryption Key (TEK) for multicast encryption/decryption and Traffic Authentication Key (TAK) for multicast authentication. Multicast messages are protected using the DTLS record layer in order to provide integrity, confidentiality and authenticity to the IP multicast messages in the LLN.

1.1. Terminology

This specification defines the following terminology:

Crypto Session ID (CS_ID): Unique identifier for a secure multicast session.

Controller: The entity that is responsible for creating a multicast group, adding members, and distributing keying material to members of the group. It is also responsible for renewing/updating the multicast group keys. It is not necessarily the sender in the multicast group.

Sender: The entity that sends multicast messages to the multicast group.

Listener: The entity that receives multicast messages when listening to a multicast IP address.

Group Security Association (GSA): A bi-directional secure channel between the controller and the member device that guarantees the confidentiality, integrity and authenticity of the messages exchanged between them.

TEK Generation Key (TGK): A bit string generated randomly and then distributed by the controller to all members of a multicast group. From the TGK, the multicast group keys (Traffic Encryption Key and Traffic Authentication Key) can then be generated.

Traffic Encryption Key (TEK): The key used to encrypt the multicast message.

Traffic Authentication Key (TAK): The key used to compute the Message Authentication Code (MAC) of the multicast message.

PRF(k,x): A keyed pseudo-random function.

||: Denotes concatenation of two bit strings.

XOR: Exclusive OR

1.2. Outline

This draft is structured as follows: Section 2 motivates the proposed solution with multicast use cases in LLNs and derives a set of requirements. Section 3 provides an overview of the DTLS-based multicast security. In Section 4, we describe the creation of a group security association (GSA) using DTLS to distribute keying materials, and the generation of group keys based on the MIKEY Architecture [RFC3830]. Section 5 proposes the use of DTLS record layer to encrypt and integrity protect multicast messages, while Section 6 discusses the group key renewal. Section 7 and Section 8 describe Security and IANA considerations.

2. Use Cases and Requirements

This section defines the use cases for multicast and specifies a set of security requirements for these use cases.

2.1. Use Cases

As stated in the Group Communication for CoAP Internet Draft [I-D.ietf-core-groupcomm] in the IETF CoRE WG, multicast is essential in several application use cases. Consider a building equipped with 6LoWPAN [RFC4944] [RFC6282] IP-connected lighting devices, switches, and 6LoWPAN border routers; the devices are organized as groups according to their location in the building, e.g., lighting devices and switches in a room/floor can be configured as a multicast group, the switches are then used to control the lighting devices in the group by sending on/off/dimming commands to the group. 6LoWPAN border routers that are connected to an IPv6 network backbone (which is also multicast enabled) are used to interconnect 6LoWPANs in the building. Consequently, this would also enable multicast groups to be formed across different subnets in the entire building. The following lists a few multicast group communication uses cases in a building management system; a detailed description of each use case can be found in Group Communication for CoAP Internet Draft [I-D.ietf-core-groupcomm].

- a. Lighting control: enabling synchronous operation of a group of 6LoWPAN connected lights in a room/floor/building. This ensures that the light preset of a large group of luminaires are changed at the same time, hence providing a visual synchronicity of light effects to the user.

- b. Firmware update: firmware of devices in a building or a campus control application are updated simultaneously, avoiding an excessive load on the LLN due to unicast firmware updates.
- c. Parameter update: settings of devices are updated simultaneously and efficiently.
- d. Commissioning of above systems: information about the devices in the local network and their capabilities can be queried and requested, e.g. by a commissioning device.

2.2. Security Requirements

The Miscellaneous CoAP Group Communication Topics Internet Draft [I-D.dijk-core-groupcomm-misc] has defined a set of security requirements for group communication in LLNs. We re-iterate and further describe those security requirements in this section with respect to the use cases as presented in Section 2.1:

- a. Multicast communication topology: We only consider a one-to-many communication topology in this draft where there is only one sender device sending multicast messages to the group. This is the simplest group communication scenario that would serve the needs of a typical LLN. For example, in the lighting control use case, the switch is the only entity that is responsible for sending control commands to a group of lighting devices. These lighting devices are actuators that do not issue commands to each other. Although in other use cases, a many-to-many multicast communication topology would be required, it is much more complex and it poses greater security challenges, therefore considered as out of scope in this draft.
- b. Establishment of a Group Security Association (GSA) [RFC3740]: A secure channel must be used to distribute keying material, multicast security policy and security parameters to members of a multicast group. A GSA must be established between the controller (which manages the multicast group and may be a different device than the sender) and the group members. The 6LoWPAN border router, a device in the 6LoWPAN, or a remote server outside the 6LoWPAN could play the role of controller for distributing keying materials. Since the keying material is used to derive subsequent group keys to protect multicast messages, it is important that it is encrypted, integrity protected and authenticated when it is distributed.
- c. Multicast security policy: All group members must use the same ciphersuite to protect the authenticity, integrity and confidentiality of multicast messages. The ciphersuite can

either be negotiated or set by the controller and then distributed to the group members. It is generally very complex and difficult to require all devices to negotiate and agree with each other on the ciphersuite to be used, it is therefore more effective that the multicast security policy is set by the controller.

- d. Multicast data group authentication: It is essential to ensure that a multicast message is originated from a member of the group. The multicast group key which is known to all group members is used to provide authenticity to the multicast messages (e.g., using a Message Authentication Code, MAC). This assumes that only the sender of the multicast group is sending the message, and that all other group members are trusted not to send nor to tamper with the multicast message. In a one-to-many communication topology, the lighting devices that serve as actuators only receive control commands from an authorized switch and do not issue commands to other lighting devices in the group.
- e. Multicast data source authentication: Source authenticity is optional. It can typically be provided using public-key cryptography in which every multicast message is signed by the sender. This requires much higher computational resources on both the sender and the receivers, thus incurring too much overhead and computational requirements on devices in LLNs. Alternatively, a lightweight broadcast authentication, i.e., TESLA [RFC4082] can be deployed, however it requires devices in the multicast group to have a trusted clock and have the ability to loosely synchronize their clocks with the sender. Consequently, given that the targeted devices have limited resources, and the need for source authenticity is not critical, it is advocated that source authenticity is made optional.
- f. Multicast data integrity: A group level integrity is required to ensure that messages have not been tampered with by attackers who are not members of the multicast group.
- g. Multicast data confidentiality: Multicast message should be encrypted, as some control commands when sent in the clear could pose privacy risks to the users.
- h. Multicast data replay protection: It must not be possible to replay a multicast message as this would disrupt the operation of the group communication.
- i. Multicast key management: Group keys used to protect the multicast communication must be renewed periodically. When members have left the multicast group, the group keys might be

leaked; and when a device is detected to have been compromised, this also implies that the group keys could have been compromised too. In these situations, the controller must perform a re-key protocol to renew the group keys.

3. Overview of DTLS-based Secure Multicast

The goal of this draft is to secure IP multicast operations as used in 6LoWPAN networks, by extending the use of the DTLS security protocol to allow for group keys distribution, and using the DTLS record layer to provide protection to multicast messages, specifically CoAP group communication. The IETF CoRE WG has selected DTLS [RFC4347] as the default must-implement security protocol for securing CoAP, therefore it is conceivable that DTLS can be extended to facilitate CoAP-based group communication. Reusing DTLS for different purposes while guaranteeing the required security properties can avoid the need to implement multiple security handshake protocols and this is especially beneficial when the target deployment consists of resource-constrained embedded devices. This section first describes group communication based on IP multicast, and subsequently sketches a solution for securing group communication using DTLS.

3.1. IP Multicast

Devices in the LLN are categorized into two roles, (1) sender and (2) listener. Any node in the LLN may have one of these roles, or both roles. The application(s) running on a device basically determine these roles by the function calls they execute on the IP stack of the device. In principle, a sender does not require any prior access procedures or authentication to send a multicast message, a sender with a valid multicast group key can essentially send a secure multicast message to the group. A device becomes a listener to a specific IP multicast group by listening to the associated IP multicast address. Any device can in principle decide to listen to any IP multicast address, and can use the associated valid group key to authenticate and decrypt the multicast messages. This also means that no prior access procedure is required to be a listener nor do applications on the other devices know, or get notified, of new listeners in the LLN.

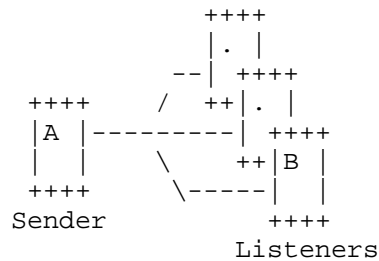


Figure 3.1: The roles of nodes in a one-to-many multicast communication topology

3.2. Securing Multicast in LLNs

A controller in an LLN creates a multicast group. The controller may be hosted by a remote server, or a border router that creates a new group over the network. In some cases, devices may be configured using a commissioning tool that mediates the communication between the devices and the controller. The controller in the network can be discovered by the devices using various methods defined in [I-D.vanderstok-core-dna] such as DNS-SD [I-D.cheshire-dnsext-dns-sd] and Resource Directory [I-D.shelby-core-resource-directory]. The controller communicates with individual device to add them to the new group. The controller establishes a GSA with each member device by performing a DTLS handshake protocol. The established DTLS secure channel (DTLS session) is then used by the controller to securely distribute over the network:

- Keying material (known as the TEK Generation Key, TKG), used for deriving multicast group keys.
- Multicast identifier, a unique identifier for the multicast group. This is typically the multicast IP address.
- Multicast security policy, which defines the ciphersuite for multicast encryption and authentication.
- Security parameters, used for generating group keys.

These parameters must be the same for all members of the group. Based on the TKG and the security parameters received, each member generates a multicast Traffic Encryption Key (TEK), and a Traffic Authentication Key (TAK) to be used for the multicast session. Each member also creates a Crypto Session (CS) to store security information (e.g., TKG, TEK, TAK, multicast identifier, ciphersuite, etc) relevant to the multicast session.

A designated sender in the group can encrypt application messages using the TEK and signs the message using the TAK. The message is then encapsulated using the DTLS record layer before it is sent using IP multicast. For example, a CoAP message addressed to a multicast group is protected using DTLS record layer and then sent to a multicast group. The listeners when receiving the message, use the multicast IP address (i.e., Multicast identifier) to look up the corresponding crypto session to obtain the TEK and TAK. The received message is decrypted using the TEK, and the authenticity is verified using the TAK.

The TEK and TAK can be renewed and updated using a re-key protocol. The controller sends new security parameters for renewing TEK and TAK over the DTLS unicast channel it has established with each group member. Using the secure unicast channels provides better reliability and security as members can individually acknowledge receipts of the new security parameters, and secondly the security parameters are protected with each member's DTLS unicast session key. One of the reasons to renew the multicast group key is that the current TEK and TAK could have been compromised, hence it defeats the purpose of the re-keying process if the controller were to distribute the new security parameters via multicast. The controller has a re-key schedule and in general the controller should update the group keys when the group membership changes.

4. Multicast Group Keys Generation and Distribution

This section describes the usage of DTLS handshake protocol to establish a GSA with all group members in order to facilitate group key distribution and management. Participating devices shall have been pre-configured with a Pre-Shared Key (PSK), raw public-key [I-D.ietf-tls-oob-pubkey] or public-key certificate, preferably individual per device. When PSK and raw public key are used, they shall also be known to the controller (through an out-of-band communication channel), so that the controller is able to authenticate and establish a secure channel with each participating device.

4.1. DTLS based Group Security Association (GSA)

The controller is commissioned to set up a multicast group. The controller performs the standard DTLS handshake protocol with each participating device in order to establish a pairwise DTLS session key. Similar to the use of DTLS in CoAP [I-D.ietf-core-coap], the DTLS handshake protocol can be performed based on PSK mode, raw public key mode or public key certificate mode. In the end, the controller establishes a DTLS security channel with each member of

the multicast group in the sense that each session is distinct from the other. The DTLS handshake protocol is shown as below:

Client		Server
	<-----	HelloRequest*
ClientHello	----->	
	<-----	HelloVerifyRequest*
ClientHello (Cookie)	----->	
		ServerHello
		Certificate*
		ServerKeyExchange*
		CertificateRequest*
	<-----	ServerHelloDone
Certificate*		
ClientKeyExchange		
CertificateVerify*		
[ChangeCipherSpec]		
Finished	----->	
		[ChangeCipherSpec]
	<-----	Finished

Figure 4.1: DTLS handshake protocol

* indicates optional messages in DTLS. When PSK is used, the ServerKeyExchange message may contain a PSK Identity hint, and the ClientKeyExchange contains a PSK identity.

Depending on the implementation, both the controller and the device may be implemented as a DTLS Client or a DTLS Server. Regardless of their roles, it is advocated that the controller initiates the DTLS handshake. When the controller implements the DTLS Client, it sends a ClientHello message to the device, otherwise it sends a HelloRequest message to initiate the DTLS handshake protocol.

The established DTLS secure channel must provide both confidentiality and integrity of the messages exchanged between the controller and the member device. Through this secure channel, the controller distributes a TEK Generation Key (TGK), a multicast security policy and security parameters to the member device over the DTLS secure channel. The TGK is generated using a pseudorandom function, and it SHALL serve as the 'master' key to derive the TEK and TAK for securing multicast communication. The TGK SHALL be at least 128-bit in length. The security parameters consist of a Multicast Identifier (Mul_ID), a Crypto Session identifier (CS_ID), and a random number (RAND). In this context, the Mul_ID is the multicast address of the group, the CS_ID is a unique identifier for the crypto session and the RAND MUST be a (at least) 128-bit pseudo-random bit string.

These parameters must be the same for all members of the multicast group. This draft defines a multicast security policy which consists of only two ciphersuites to protect multicast messages. All member devices must support the following ciphersuites:

```
Ciphersuite MTS_WITH_AES_128_CCM_8 = {TBD1, TBD2}
Ciphersuite MTS_WITH_NULL_SHA256   = {TBD3, TBD4}
```

Ciphersuite MTS_WITH_AES_128_CCM_8 is used to provide confidentiality, integrity and authenticity to the multicast messages where the encryption algorithm is AES [AES], key length is 128-bit, and the authentication function is CCM [RFC6655] with a Message Authentication Code (MAC) length of 8 bytes. Similar to [RFC4785], the ciphersuite MTS_WITH_NULL_SHA is used when confidentiality of multicast messages is not required, it only provides integrity and authenticity protection to the multicast message. When this ciphersuite is used, the message is not encrypted but the MAC must be included in which it is computed using a HMAC [RFC2104] that is based on Secure Hash Function (SHA256) [SHA]. Depending on the future needs, other ciphersuites with different cipher algorithms and MAC length may be supported.

The GSA (i.e., the DTLS secure channel) established is kept to facilitate group key renewals, thus allowing the controller to distribute new security parameters to members of the multicast group to update the group keys. This is further described in Section 6.

4.2. Generation of Group Keys

Once the member device has received the security parameters, multicast security policy and the TGK from the controller, the device generates the Traffic Encryption Key (TEK) and Traffic Authentication Key (TAK) using the Pseudo Random Function (PRF) as defined in Section 4.1 in MIKEY [RFC3830]. The TEK is used as the common group key known to all members of the group to encrypt multicast messages, while the TAK is used to create a MAC for the message. The DTLS record layer advocates the use of different key for encryption and authentication.

Similar to MIKEY [RFC3830], the following input parameters are defined:

```
inkey       : the input key to the key generation function.
inkey_len   : the length in bits of the input key.
label       : a specific label, dependent on the type of the key to be
               generated, the random number, and the session IDs.
outkey_len  : desired length in bits of the output key.
```

The key generation function has the following output:

outkey: the output key of desired length.

The following defines the input parameters to the group keys generation function. These input parameters are distributed by the controller and used by the devices in a multicast group to generate group keys.

```
inkey       : TGK
inkey_len   : bit length of TGK
label       : constant || mul_id || cs_id || RAND
outkey_len  : bit length of the output key.
```

As defined in MIKEY [RFC3830], the constant part of label depends on the type of key that is to be generated. The constant 0x2AD01C64 is used to generate a TEK from TGK, while the the constant 0x1B5C7973 is used to generate a TAK. The outkey_len SHALL be set to 128 bit. A crypto session should be created to store information about the multicast session, providing a mapping of the multicast identifier to the TEK, TAK, the security parameters and the multicast security policy as well as the information about the controller that is associated with the multicast session.

The following re-iterates the key generation procedure as described in MIKEY [RFC3830] with the difference that SHA256 is used instead of SHA-1.

The PRF(inkey,label) that is based on the P-function in MIKEY [RFC3830] is applied to compute the output keys (TEK and TAK):

- o Let $n = \text{inkey_len} / 256$, rounded up to the nearest integer if not already an integer
- o Split the inkey into n blocks, $\text{inkey} = s_1 || \dots || s_n$, where all s_i , except possibly s_n , are 256 bits each
- o Let $m = \text{outkey_len} / 256$, rounded up to the nearest integer if not already an integer

(The values "256" equal half the input block-size and full output hash size of the SHA256 as part of the P-function.)

Then, the output key, outkey, is obtained as the outkey_len most significant bits of

```
PRF(inkey, label) = P(s_1, label, m) XOR P(s_2, label, m) XOR ...
XOR P(s_n, label, m).
```


5. Multicast Data Security

This section describes the use of DTLS record layer to secure multicast messages.

5.1. Sending Secure Multicast Messages

All messages addressed to the multicast group must be secured using the TEK and TAK. Using the DTLS record layer, multicast messages are encrypted using the TEK and a Message Authentication Code (MAC) is generated using the TAK according to the ciphersuite defined in the multicast security policy. The MAC is appended to the encrypted message before it is passed down to the lower layer of the IP protocol stack for transmission to the multicast address.

As described in Section 4.1, the ciphersuite `MTS_WITH_AES_128_CCM_8` defines that the multicast message must be encrypted using AES with a 128-bit TEK. Since the CCM mode of operation is used for authenticated encryption, the same TEK is used to compute the MAC and the TAK is not used. As for the ciphersuite `MTS_WITH_NULL_SHA`, the multicast message must not be encrypted, but a MAC must be computed using the TAK key.

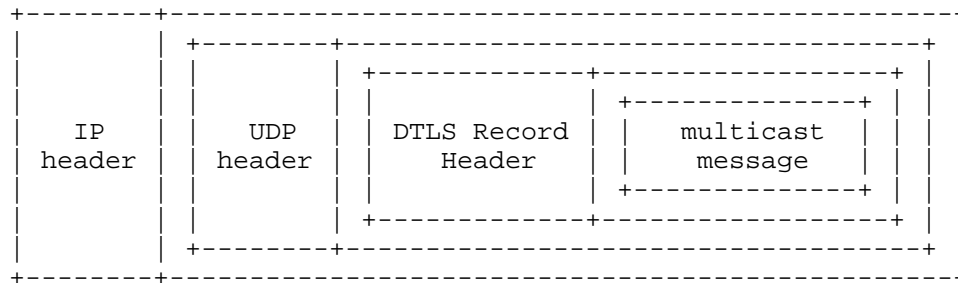


Figure 5.1: Sending a multicast message protected using DTLS Record Layer

The DTLS record layer header contains a 48-bit sequence number that is used for (1) allowing the recipient to correctly verify the DTLS MAC, (2) preventing message replay. The current use of the sequence number is adequate in a one-to-many multicast communication topology. The sequence number is generated by the sender as specified in DTLS. The sequence number field in the DTLS record layer header is incremented whenever the sender sends a multicast message. This requires all member devices to keep track of the sequence number received, so that the message freshness can be verified.

5.2. Receiving Secure Multicast Messages

Member devices receiving the multicast message, look up the crypto session to find the corresponding TEK and TAK to decrypt and verify the MAC of the multicast message. The destination multicast IP address which serves as the Multicast identifier (Mul_ID) can be used to locate the crypto session in order to obtain the TEK and TAK. The crypto session must also contain the last received message's epoch and sequence number, enabling the member devices to detect message replay. Multicast messages received with a sequence number less than or equal to the value stored in the crypto session must be dropped. The epoch number in the received message must also match the epoch number stored in the corresponding crypto session. As a consequence of this mechanism, a message that arrives out-of-order (i.e. with a sequence number less than the value stored in the crypto session) will be ignored.

This replay detection mechanism only applies to one-to-many communication topology, where member devices are assumed to be trusted not to tamper with the messages.

6. Group Keys Renewal

The controller can initiate re-key of the TEK and TAK according to a key renewal schedule and when the group membership changes. It is important that the group keys, i.e., TEK and TAK are renewed periodically to prevent potential attacks and cryptanalysis. When performing re-key, the controller generates a new Random number (RAND), and a new crypto session ID (CS_ID), and subsequently sends this information through the unicast DTLS secure channel established with each member. The new TEK and TAK are then generated by each member based on the algorithm described in Section 4.2, using the new RAND and CS_ID received from the controller. The TGK which serves as the 'master' group key does not change. When the TEK and TAK have been updated, the epoch number maintained in the multicast crypto session must be incremented.

7. IANA Considerations

tbd

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

8. Security Considerations

tbd

9. Acknowledgements

The authors greatly acknowledge discussion, comments and feedback from Dee Denteneer, Peter van der Stok and Zach Shelby. We also appreciate prototyping and implementation efforts by Pedro Moreno Sanchez who works as an intern at Philips Research.

10. References

10.1. Normative References

- [AES] National Institute of Standards and Technology, "Specification for the Advanced Encryption Standard (AES)", FIPS 197, Nov 2001.
- [RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC3740] Hardjono, T. and B. Weis, "The Multicast Group Security Architecture", RFC 3740, March 2004.
- [RFC3830] Arkko, J., Carrara, E., Lindholm, F., Naslund, M., and K. Norrman, "MIKEY: Multimedia Internet KEYing", RFC 3830, August 2004.
- [RFC4347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security", RFC 4347, April 2006.
- [RFC6655] McGrew, D. and D. Bailey, "AES-CCM Cipher Suites for Transport Layer Security (TLS)", RFC 6655, July 2012.
- [SHA] National Institute of Standards and Technology, "Secure Hash Standard", FIPS 180-2, Aug 2002.

10.2. Informative References

- [I-D.cheshire-dnsext-dns-sd] Cheshire, S. and M. Krochmal, "DNS-Based Service

Discovery", draft-cheshire-dnsext-dns-sd-11 (work in progress), December 2011.

[I-D.dijk-core-groupcomm-misc]

Dijk, E. and A. Rahman, "Miscellaneous CoAP Group Communication Topics", draft-dijk-core-groupcomm-misc-01 (work in progress), July 2012.

[I-D.ietf-core-coap]

Shelby, Z., Hartke, K., Bormann, C., and B. Frank, "Constrained Application Protocol (CoAP)", draft-ietf-core-coap-12 (work in progress), October 2012.

[I-D.ietf-core-groupcomm]

Rahman, A. and E. Dijk, "Group Communication for CoAP", draft-ietf-core-groupcomm-02 (work in progress), July 2012.

[I-D.ietf-tls-oob-pubkey]

Wouters, P., Gilmore, J., Weiler, S., Kivinen, T., and H. Tschofenig, "Out-of-Band Public Key Validation for Transport Layer Security", draft-ietf-tls-oob-pubkey-04 (work in progress), July 2012.

[I-D.shelby-core-resource-directory]

Shelby, Z., Krco, S., and C. Bormann, "CoRE Resource Directory", draft-shelby-core-resource-directory-04 (work in progress), July 2012.

[I-D.vanderstok-core-dna]

Stok, P., Lynn, K., and A. Brandt, "CoRE Discovery, Naming, and Addressing", draft-vanderstok-core-dna-02 (work in progress), July 2012.

[RFC4082] Perrig, A., Song, D., Canetti, R., Tygar, J., and B. Briscoe, "Timed Efficient Stream Loss-Tolerant Authentication (TESLA): Multicast Source Authentication Transform Introduction", RFC 4082, June 2005.

[RFC4785] Blumenthal, U. and P. Goel, "Pre-Shared Key (PSK) Ciphersuites with NULL Encryption for Transport Layer Security (TLS)", RFC 4785, January 2007.

[RFC4944] Montenegro, G., Kushalnagar, N., Hui, J., and D. Culler, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks", RFC 4944, September 2007.

[RFC6282] Hui, J. and P. Thubert, "Compression Format for IPv6

Datagrams over IEEE 802.15.4-Based Networks", RFC 6282,
September 2011.

Authors' Addresses

Sye Loong Keoh
Philips Research
High Tech Campus 34
Eindhoven 5656 AE
NL

Email: sye.loong.keoh@philips.com

Oscar Garcia Morchon
Philips Research
High Tech Campus 34
Eindhoven 5656 AE
NL

Email: oscar.garcia@philips.com

Sandeep S. Kumar
Philips Research
High Tech Campus 34
Eindhoven 5656 AE
NL

Email: sandeep.kumar@philips.com

Esko Dijk
Philips Research
High Tech Campus 34
Eindhoven 5656 AE
NL

Email: esko.dijk@philips.com

TLS Working Group
Internet-Draft
Intended status: Standards Track
Expires: March 30, 2013

M. Marlinspike
T. Perrin, Ed.
September 26, 2012

Trust Assertions for Certificate Keys
draft-perrin-tls-tack-01.txt

Abstract

This document defines TACK, a TLS Extension that enables a TLS server to assert the authenticity of its public key. A "tack" contains a "TACK key" which is used to sign the public key from the TLS server's certificate. Hostnames can be "pinned" to a TACK key. TLS connections to a pinned hostname require the server to present a tack containing the pinned key and a corresponding signature over the TLS server's public key.

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 March 30, 2013.

Copyright Notice

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

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

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

Table of Contents

1. Introduction	3
2. Requirements notation	4
3. Overview	5
3.1. Tack life cycle	5
3.2. Pin life cycle	6
4. TACK Extension	7
4.1. Definition of TackExtension	7
4.2. Explanation of TackExtension fields	8
4.2.1. Tack fields	8
4.2.2. TackExtension fields	8
5. Client processing	9
5.1. TACK pins	9
5.2. High-level client processing	9
5.3. Client processing details	10
5.3.1. Check whether the TLS handshake is well-formed	10
5.3.2. Check tack generations and update min_generations	10
5.3.3. Determine the store's status	11
5.3.4. Pin activation (optional)	11
6. Application protocols and TACK	13
6.1. Pin scope	13
6.2. TLS negotiation	13
6.3. Certificate verification	13
7. Fingerprints	14
8. Advice	15
8.1. For server operators	15
8.2. For client implementers	16
9. Security considerations	17
9.1. For server operators	17
9.2. For client implementers	17
9.3. Note on algorithm agility	18
10. IANA considerations	19
10.1. New entry for the TLS ExtensionType Registry	19
11. Acknowledgements	20
12. Normative references	21
Authors' Addresses	22

1. Introduction

Traditionally, a TLS client verifies a TLS server's public key using a certificate chain issued by some public CA. "Pinning" is a way for clients to obtain increased certainty in server public keys. Clients that employ pinning check for some constant "pinned" element of the TLS connection when contacting a particular TLS host.

Unfortunately, a number of problems arise when attempting to pin certificate chains: the TLS servers at a given hostname may have different certificate chains simultaneously deployed and may change their chains at any time, the "more constant" elements of a chain (the CAs) may not be trustworthy, and the client may be oblivious to key compromise events which render the pinned data untrustworthy.

TACK addresses these problems by having the site sign its TLS server public keys with a "TACK key". This enables clients to "pin" a hostname to the TACK key without requiring sites to modify their existing certificate chains, and without limiting a site's flexibility to deploy different certificate chains on different servers or change certificate chains at any time. Since TACK pins are based on TACK keys (instead of CA keys), trust in CAs is not required. Additionally, the TACK key may be used to revoke compromised TLS private keys, and TACK key rollovers may be performed to recover from suspect or compromised TACK keys.

If requested, a compliant server will send a TLS Extension containing its "tack". Inside the tack is a public key and signature. Once a client has seen the same (hostname, TACK public key) pair multiple times, the client will "activate" a pin between the hostname and TACK key for a period equal to the length of time the pair has been observed for. This "pin activation" algorithm limits the impact of bad pins resulting from transient network attacks or operator error.

TACK pins are easily shared between clients. For example, a TACK client may scan the internet to discover TACK pins, then publish these pins through some 3rd-party trust infrastructure for other clients to rely upon.

2. Requirements notation

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 [RFC2119].

3. Overview

3.1. Tack life cycle

A server operator using TACK may perform several processes:

Selection of a TACK key: The server operator first chooses the ECDSA signing key to use for a set of hostnames. It is safest to use a different signing key for each hostname, though a signing key may be reused for closely-related hostnames (such as aliases for the same host, or hosts sharing the same TLS key).

Creating initial tacks under a TACK key: The TACK private key is then used to sign the TLS public keys for all servers associated with those hostnames. The TACK public key and signature are combined with some metadata into each server's "tack".

Deploying initial tacks: For each hostname, tacks are deployed to TLS servers in a two-stage process. First, each TLS server associated with the hostname is given a tack. Once this is completed, the tacks are activated by setting the "activation flag" on each server.

Creating new tacks under a TACK key: A tack needs to be replaced whenever a server changes its TLS public key, or when the tack expires. Tacks may also need to be replaced with later-generation tacks if the TACK key's "min_generation" is updated (see next).

Revoking old tacks: If a TLS private key is compromised, the tacks signing this key can be revoked by publishing a new tack containing a higher "min_generation".

Deactivating tacks: If a server operator wishes to stop deploying tacks, all tacks for a hostname can be deactivated via the activation flag, allowing the server to remove the tacks within 30 days (at most).

Rollover: If a server operator wishes to change the TACK key a hostname is pinned to, the server can publish a new tack alongside the old one. This lets clients activate pins for the new TACK key prior to the server deactivating the older pins.

3.2. Pin life cycle

A TACK pin associates a hostname and a TACK key. Pins are grouped into "pin stores". A client may populate its pin stores by either performing "pin activation" directly, or by querying some other party. For example, a client application may have a store for pin activation as well as a store whose contents are periodically fetched from a server.

Whenever a client performing "pin activation" sees a hostname and TACK key combination not represented in the "pin activation" pin store, an inactive pin is created. Every subsequent time the client sees the same pin, the pin is "activated" for a period equal to the timespan between the first time the pin was seen and the most recent time, up to a maximum period of 30 days.

A pin store may contain up to two pins per hostname. This allows for "pin rollover", where a server securely transitions from one pin to another. If both pins are simultaneously active, then the server must satisfy both of them by presenting a pair of tacks.

In addition to creating and activating pins, a TLS connection can alter client pin stores by publishing new "min_generation" values in a tack. Each pin stores the highest "min_generation" value it has seen from the pinned TACK key, and rejects tacks from earlier generations.

4. TACK Extension

4.1. Definition of TackExtension

A new TLS ExtensionType ("tack") is defined and MAY be included by a TLS client in the ClientHello message defined in [RFC5246].

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

The "extension_data" field of this ClientHello extension SHALL be empty. A TLS server which is not resuming a TLS session MAY respond with an extension of type "tack" in the ServerHello. The "extension_data" field of this ServerHello extension SHALL contain a "TackExtension", as defined below using the TLS presentation language from [RFC5246].

```
struct {
    opaque public_key[64];
    uint8  min_generation;
    uint8  generation;
    uint32 expiration;
    opaque target_hash[32];
    opaque signature[64];
} Tack; /* 166 bytes */

struct {
    Tack  tacks<166...332> /* 1 or 2 Tacks */
    uint8 activation_flags; /* 0...3 */
} TackExtension;
```

4.2. Explanation of TackExtension fields

4.2.1. Tack fields

public_key: This field specifies the tack's public key. The field contains a pair of integers (x, y) representing a point on the elliptic curve P-256 defined in [FIPS186-3]. Each integer is encoded as a 32-byte octet string using the Integer-to-Octet-String algorithm from [RFC6090], and these strings are concatenated with the x value first. (NOTE: This is equivalent to an uncompressed subjectPublicKey from [RFC5480], except that the initial 0x04 byte is omitted).

min_generation: This field publishes a min_generation value.

generation: This field assigns each tack a generation. Generations less than a published min_generation are considered revoked.

expiration: This field specifies a time after which the tack is considered expired. The time is encoded as the number of minutes, excluding leap seconds, after midnight UTC, January 1 1970.

target_hash: This field is a hash of the TLS server's SubjectPublicKeyInfo [RFC5280] using the SHA256 algorithm from [FIPS180-2]. The SubjectPublicKeyInfo is typically conveyed as part of the server's X.509 certificate.

signature: This field is an ECDSA signature by the tack's public key over the 8 byte ASCII string "tack_sig" followed by the contents of the tack prior to the "signature" field (i.e. the preceding 102 bytes). The field contains a pair of integers (r, s) representing an ECDSA signature as defined in [FIPS186-3], using curve P-256 and SHA256. Each integer is encoded as a 32-byte octet string using the Integer-to-Octet-String algorithm from [RFC6090], and these strings are concatenated with the r value first.

4.2.2. TackExtension fields

tacks: This field provides the server's tack(s). It SHALL contain 1 or 2 tacks.

activation_flags: This field contains "activation flags" for the extension's tacks. If the low order bit is set, the first tack is considered active. If the next lowest bit is set, the second tack is considered active. An active tack MAY be used by the pin activation algorithm in Section 5.3.4 to create, activate, and extend the activation of TACK pins.

5. Client processing

5.1. TACK pins

A client SHALL have a local store of pins, and MAY have multiple stores. Each pin store consists of a map associating fully qualified DNS hostnames with either one or two sets of the following values:

Initial time: A timestamp noting when this pin was created.

End time: A timestamp determining the pin's "active period". If set to zero or a time in the past, the pin is "inactive". If set to a future time, the pin is "active" until that time.

TACK public key (or hash): A public key or a cryptographically-secure, second preimage-resistant hash of a public key.

Min_generation: A single byte used to detect revoked tacks. All pins within a pin store sharing the same TACK public key SHALL have the same min_generation.

A hostname along with the above values comprises a "TACK pin". Thus, each store can hold up to two pins for a hostname (however, those two pins MUST reference different public keys). A pin "matches" a tack if they reference the same public key. A pin is "relevant" if its hostname equals the TLS server's hostname.

5.2. High-level client processing

A TACK client SHALL send the "tack" extension defined previously, and SHALL send the "server_name" extension from [RFC6066]. If not resuming a session, the server MAY respond with a TackExtension. Regardless of whether a TackExtension is returned, the client SHALL perform the following steps prior to using the connection:

1. Check whether the TLS handshake is "well-formed".
2. For each pin store, do:
 - A. Check tack generations and update min_generations.
 - B. Determine the store's status.
 - C. Perform pin activation (optional).

These steps SHALL be performed in order. If there is any error, the client SHALL send a fatal error alert and close the connection, skipping the remaining steps (see Section 5.3 for details).

Based on step 2B, each store will report one of three statuses for the connection: "accepted", "rejected", or "unpinned". A rejected connection might indicate a network attack. If the connection is rejected the client SHOULD send a fatal "access_denied" error alert and close the connection.

A client MAY perform additional verification steps before using an accepted or unpinned connection. See Section 6.3 for an example.

5.3. Client processing details

5.3.1. Check whether the TLS handshake is well-formed

A TLS handshake is "well-formed" if the following are true. Unless otherwise specified, if any of the following are false a "bad_certificate" fatal error alert SHALL be sent.

1. The handshake protocol negotiates a cryptographically secure ciphersuite and finishes successfully.
2. If a TackExtension is present then all length fields are correct, "activation_flags" is ≤ 3 , and the tacks are "well-formed" (see below).
3. If there are two tacks, they have different "public_key" fields.

A tack is "well-formed" if:

1. "generation" is \geq "min_generation".
2. "expiration" specifies a time in the future, otherwise the client SHALL send a fatal "certificate_expired" error alert.
3. "target_hash" is a correct hash of the SubjectPublicKeyInfo.
4. "signature" is a correct ECDSA signature.

5.3.2. Check tack generations and update min_generations

If a tack has matching pins in the pin store and a generation less than the stored min_generation, then that tack is revoked and the client SHALL send a fatal "certificate_revoked" error alert. If a tack has matching pins and a min_generation greater than the stored min_generation, the stored value SHALL be set to the tack's value.

5.3.3. Determine the store's status

If there is a relevant active pin without a matching tack, then the connection is "rejected". If the connection is not rejected and there is a relevant active pin with a matching tack, then the connection is "accepted". Otherwise, the connection is "unpinned".

5.3.4. Pin activation (optional)

The TLS connection MAY be used to create, delete, and activate pins. This "pin activation algorithm" is optional; a client MAY rely on an external source of pins. If the connection was "rejected" by the previous processing step, then pin activation is skipped.

The first step in pin activation is to evaluate each relevant pin (there may be one or two):

1. If a pin has no matching tack, its handling will depend on whether the pin is active. If active, the connection will have been rejected, skipping pin activation. If inactive, the pin SHALL be deleted, since it is contradicted by the connection.
2. If a pin has a matching tack, its handling will depend on whether the tack is active. If inactive, the pin is left unchanged. If active, the pin SHALL have its "end time" set based on the current, initial, and end times:

end = current + MIN(30 days, current - initial)

In sum: (1) deletes unmatched pins, provided they are inactive; and (2) activates matched pins, provided the matching tack is active.

The remaining step in pin activation is to add new inactive pins for any unmatched active tacks. Each new pin uses the server's hostname, the tack's public key and min_generation (unless the store has a higher min_generation for the public key), an "initial time" set to the current time, and an "end time" of zero.

(Note that there are always sufficient empty "slots" in the pin store for adding new pins without exceeding two pins per hostname. This is because the number of matching pins equals the number of matching tacks, so the number of empty pin slots equals the number of unmatched tacks.)

The following tables summarize this behavior from the perspective of a pin. You can follow the lifecycle of a single pin from "New inactive pin" to "Delete pin".

Relevant pin is active:

Pin matches a tack	Tack is active	Result
Yes	Yes	Extend activation period
Yes	No	-
No	-	(Connection rejected)

Relevant pin is inactive:

Pin matches a tack	Tack is active	Result
Yes	Yes	Activate pin
Yes	No	-
No	-	Delete pin

Tack doesn't match any relevant pin:

Unmatched tack is active	Result
Yes	New inactive pin
No	-

6. Application protocols and TACK

6.1. Pin scope

TACK pins are specific to a particular application protocol. In other words, a pin for HTTPS at "example.com" implies nothing about POP3 or SMTP at "example.com".

6.2. TLS negotiation

Some application protocols negotiate TLS as an optional feature (e.g. SMTP using STARTTLS [RFC3207]). If such a server fails to negotiate TLS and there are relevant active pins, then the connection is rejected by the pin. If the server fails to negotiate TLS, then any relevant, inactive pins SHALL be deleted. Note that these steps are taken despite the absence of a TLS connection.

6.3. Certificate verification

A TACK client MAY choose to perform some form of certificate verification in addition to TACK processing. When combining certificate verification and TACK processing, the TACK processing described in Section 5 SHALL be followed, with the exception that TACK processing MAY be terminated early (or skipped) if some fatal certificate error is discovered.

If TACK processing and certificate verification both complete without a fatal error, the client SHALL apply some policy to decide whether to accept the connection. The policy is up to the client. An example policy would be to accept the connection only if it passes certificate verification and is not rejected by a pin.

7. Fingerprints

A "key fingerprint" may be used to represent a TACK public key to users in a form that is easy to compare and transcribe. A key fingerprint consists of the first 25 characters from the base32 encoding of SHA256(public_key), split into 5 groups of 5 characters separated by periods. Base32 encoding is as specified in [RFC4648], except lowercase is used. Examples:

g5p5x.ov4vi.dgsjv.wxctt.c5iul

quxiz.kpldu.uuedc.j5znm.7mqst

e25zs.cth7k.tscmp.5hxdp.wf47j

8. Advice

8.1. For server operators

Key reuse: All servers that are pinned to a single TACK key are able to impersonate each other, since clients will perceive their tacks as equivalent. Thus, TACK keys SHOULD NOT be reused with different hostnames unless these hostnames are closely related. Examples where it would be safe to reuse a TACK key are hostnames aliased to the same host, hosts sharing the same TLS key, or hostnames for a group of near-identical servers.

Aliases: A TLS server may be referenced by multiple hostnames. Clients may pin any of these hostnames. Server operators should be careful when using DNS aliases that hostnames are not pinned inadvertently.

Generations: To revoke older generations of tacks, the server operator SHOULD first provide all servers with a new generation of tacks, and only then provide servers with new tacks containing the new `min_generation`. Otherwise, a client may receive a `min_generation` update from one server but then try to contact an older-generation server which has not yet been updated.

Tack expiration: When TACK is used in conjunction with certificates it is recommended to set the tack expiration equal to the end-entity certificate expiration plus 30 days, allowing the tack and certificate to both be replaced at the same time. The extra 30 days ensures there is enough time to employ "pin deactivation" (see below) if the TACK private key is lost. Alternatively, short-lived tacks may be used so that a compromised TLS private key has limited value to an attacker.

Tack/pin activation: Tacks should only be activated once all TLS servers sharing the same hostname have a tack. Otherwise, a client may activate a pin by contacting one server, then contact a different server at the same hostname that does not yet have a tack.

Tack/pin deactivation: If all servers at a hostname deactivate their tacks (by clearing the activation flags), all existing pins for the hostname will eventually become inactive. The tacks can be removed after a time interval equal to the maximum active period of any affected pins (30 days at most).

Pin rollover: When performing a rollover, the old and new tacks SHOULD be published simultaneously for at least 60 days. This ensures that a pin activation client who is contacting the server at least once every 30 days will not have the length of its activation periods affected by the transition. Example rollover process: Add new tacks; activate new tacks; wait 30+ days; deactivate old tacks; wait 30+ days; remove old tacks.

8.2. For client implementers

Sharing pin information: It is possible for a client to maintain a pin store based entirely on its own TLS connections. However, such a client runs the risk of creating incorrect pins, failing to keep its pins active, or failing to receive `min_generation` updates. Clients are advised to make use of 3rd-party trust infrastructure so that pin data can be aggregated and shared. This will require additional protocols outside the scope of this document.

Clock synchronization: A client SHOULD take measures to prevent tacks from being erroneously rejected as expired due to an inaccurate client clock. Such methods MAY include using time synchronization protocols such as NTP [RFC5905], or accepting seemingly-expired tacks as "well-formed" if they expired less than T minutes ago, where T is a "tolerance bound" set to the client's maximum expected clock error.

9. Security considerations

9.1. For server operators

All servers pinned to the same TACK key can impersonate each other (see Section 8.1). Think carefully about this risk if using the same TACK key for multiple hostnames.

Make backup copies of the TACK private key and keep all copies in secure locations where they can't be compromised.

A TACK private key MUST NOT be used to perform any non-TACK cryptographic operations. For example, using a TACK key for email encryption, code-signing, or any other purpose MUST NOT be done.

HTTP cookies [RFC6265] set by a pinned host can be stolen by a network attacker who can forge web and DNS responses so as to cause a client to send the cookies to a phony subdomain of the pinned host. To prevent this, TACK HTTPS Servers SHOULD set the "secure" attribute and omit the "domain" attribute on all security-sensitive cookies, such as session cookies. These settings tell the browser that the cookie should only be presented back to the originating host (not its subdomains), and should only be sent over HTTPS (not HTTP) [RFC6265].

9.2. For client implementers

A TACK pin store may contain private details of the client's connection history. An attacker may be able to access this information by hacking or stealing the client. Some information about the client's connection history could also be gleaned by observing whether the client accepts or rejects connections to phony TLS servers without correct tacks. To mitigate these risks, a TACK client SHOULD allow the user to edit or clear the pin store.

Aside from rejecting TLS connections, clients SHOULD NOT take any actions which would reveal to a network observer the state of the client's pin store, as this would allow an attacker to know in advance whether a "man-in-the-middle" attack on a particular TLS connection will succeed or be detected.

An attacker may attempt to flood a client with spurious tacks for different hostnames, causing the client to delete old pins to make space for new ones. To defend against this, clients SHOULD NOT delete active pins to make space for new pins. Clients instead SHOULD delete inactive pins. If there are no inactive pins to delete, then the pin store is full and there is no space for new pins. To select an inactive pin for deletion, the client SHOULD delete the pin with the oldest "end time".

9.3. Note on algorithm agility

If the need arises for tacks using different cryptographic algorithms (e.g., if SHA256 or ECDSA are shown to be weak), a "v2" version of tacks could be defined, requiring assignment of a new TLS Extension number. Tacks as defined in this document would then be known as "v1" tacks.

10. IANA considerations

10.1. New entry for the TLS ExtensionType Registry

IANA is requested to add an entry to the existing TLS ExtensionType registry, defined in [RFC5246], for "tack"(TBD) as defined in this document.

11. Acknowledgements

Valuable feedback has been provided by Adam Langley, Chris Palmer, Nate Lawson, and Joseph Bonneau.

12. Normative references

- [FIPS180-2] National Institute of Standards and Technology, "Secure Hash Standard", FIPS PUB 180-2, August 2002, <<http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf>>.
- [FIPS186-3] National Institute of Standards and Technology, "Digital Signature Standard", FIPS PUB 186-3, June 2009, <http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC3207] Hoffman, P., "SMTP Service Extension for Secure SMTP over Transport Layer Security", RFC 3207, February 2002.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.
- [RFC5480] Turner, S., Brown, D., Yiu, K., Housley, R., and T. Polk, "Elliptic Curve Cryptography Subject Public Key Information", RFC 5480, March 2009.
- [RFC5905] Mills, D., Martin, J., Burbank, J., and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification", RFC 5905, June 2010.
- [RFC6066] Eastlake, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, January 2011.
- [RFC6090] McGrew, D., Igoe, K., and M. Salter, "Fundamental Elliptic Curve Cryptography Algorithms", RFC 6090, February 2011.
- [RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, April 2011.

Authors' Addresses

Moxie Marlinspike

Trevor Perrin (editor)

Email: tack@trevp.net

