TLS Working Group
Internet Draft

I. Hajjeh
INEOVATION
M. Badra
LIMOS Laboratory

Intended status: Experimental
Expires: May 2010

November 14, 2009

Credential Protection Ciphersuites for Transport Layer Security (TLS)
draft-hajjeh-tls-identity-protection-09.txt

Status of this Memo

Copyright Notice

Abstract

This document defines a set of cipher suites to add client credential protection to the Transport Layer Security (TLS) protocol. By negotiating one of those ciphersuites, the TLS clients will be able to determine for themselves when, how, to what extent and for what purpose information about them is communicated to others. The ciphersuites defined in this document can be used only when public key certificates are used in the client authentication process.

Table of Contents

1. Introduction

The Transport Layer Security (TLS) protocol (TLS v1.0 [RFC2246], TLS v1.1 [RFC4346], and TLS v1.2 [RFC5246]), is the most deployed security protocol for securing exchanges. It provides end-to-end secure communications between two entities with authentication and data protection.

TLS supports three authentication modes: authentication of both parties, only server-side authentication, and anonymous key exchange. For each mode, TLS specifies a set of cipher suites. However, anonymous cipher suites are strongly discouraged because they cannot prevent man-in-the-middle (MITM) attacks.

The TLS authentication is usually based on either preshared keys or public key certificates. If a public key certificate is used to authenticate the TLS client, the TLS client credentials are sent in clear text over the wire. Thus, any observer can determine the credentials used by the client; learn who is reaching the network, when, and from where, and hence correlate the client credentials to the connection location.

Credentials protection and privacy are the right to informational self-determination, i.e., individuals must be able to determine for themselves when, how, to what extent and for what purpose information about them is communicated to others.

TLS client credential protection may be established by changing the order of the messages that the client sends after receiving ServerHelloDone [CORELLA]. It consists of sending the ChangeCipherSpec message before the Certificate and the CertificateVerify messages and after the ClientKeyExchange message. The ChangeCipherSpec message is sent to notify the receiving party that subsequent messages will be protected under the cipher suite and keys negotiated during the TLS Handshake. However, this solution requires a major change to the TLS state machine as well as a new TLS version.

TLS client credential protection may also be done through a DHE exchange before establishing an ordinary handshake with identity information [SSLTLS]. This wouldn't however be secure enough against active attackers, which will be able to disclose the client's credentials. Moreover, it wouldn't be favorable for some environments (e.g., performance-constrained environments with limited CPU power), due to the additional cryptographic computations and round trips.

TLS client credential protection may also be possible, assuming that the client permits renegotiation after the first server authentication [RFC5246]: The client and the server establish a TLS session with only server-side authentication and then perform a new full TLS Handshake with mutual authentication; the client credentials transferred in this stage thus are protected by the secure channel established in the first TLS Handshake. This solution doesn't require a change to TLS. However, this solution requires more asymmetric cryptographic computations, which in many environments (in particular for less powerful mobile nodes) are the rate limiting step in TLS, and therefore, the renegotiation has negative performance consequences. In fact, renegotiation requires another round of an asymmetric encryption/decryption, which means the double number of asymmetric en-/decryption operations (e.g., with an RSA key) for TLS Handshake message processing, for both server and client. Moreover, renegotiation requires twice the number of messages and roundtrips than a single TLS handshake, thus significantly increasing the overall delay in the session setup. Additionally, the server is forced to complete a full first TLS handshake before it becomes able to confirm whether the client has a valid certificate or not. This increased misbalance in processing load in the failure case might open an opportunity for misbehaving clients to perform resource exhaustion attacks against such servers.

TLS client credential protection may as well be done by allowing the client and the server to add a TLS extension to their Hello messages in order to negotiate specific crypto algorithms, and use these to protect the client certificate [EAPIP]. This solution may suffer from interoperability issues related to TLS Extensions, TLS 1.0 and TLS 1.1 implementations, as described in [INTEROP]. Moreover, it provides imperfect privacy guarantees. In fact, the CertificateVerify message is sent in clear text over the wire. As a consequence, if an attacker ever obtains a client's certificate it can do trial verification to determine whether a new handshake uses that certificate.

This document defines a set of cipher suites to add client credential protection to the TLS protocol. When one of the cipher suites defined through this document is negotiated, a symmetric encryption is used to encrypt the TLS client Certificate and the CertificateVerify messages as following:

- The keys for the symmetric encryption and MAC are generated uniquely for each TLS Handshake and are based on a secret negotiated during the TLS Handshake. These keys don't replace the other keys and secrets (master_secret and key_block).

- Each encrypted message includes a message integrity check using a keyed MAC. Secure hash functions (e.g., SHA, etc.) are used for MAC computations.

- The encryption and MAC algorithms are determined by the cipher_suite selected by the server and revealed in the ServerHello message.

- Any key generated by this document should be deleted from memory once the CertificateVerify message has been encrypted or decrypted.

The reader is expected to become familiar with the TLS standards ([RFC5246] and, if needed, [RFC4346] and [RFC2246] for its predecessors) prior to studying this document.

1.1. Conventions used in this document

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].

2. TLS credential protection overview

This document specifies a set of cipher suites for TLS. These cipher suites reuse existing key exchange algorithms with certificate-based authentication, and reuse existing cipher and MAC algorithms from [RFC5246], [RFC4492], and [RFC4132].

The name of cipher suites defined in this document includes the text "CP" to refer to the client credential protection. An example is shown below.

CipherSuite	Key Exchange	Cipher	Hash
TLS_CP_RSA_WITH_AES_128_CBC_SHA	RSA	AES_128_CBC	SHA1
TLS_CP_DHE_DSS_WITH_AES_128_CBC_SHA	DHE	AES_128_CBC	SHA1

If no certificates are available, the client MUST NOT include any credential protection cipher suite in the ClientHello.cipher_suites.

If the server selects a cipher suite with client credential protection, the server MUST send a certificate appropriate for the negotiated cipher suite's key exchange algorithm, and MUST request a certificate from the client. If the server, agreeing on using a credential protection cipher suite, does not receive a client certificate in response to the subsequent certificate request, then it MUST abort the session by sending a fatal handshake failure alert.

The client certificate MUST be appropriate for the negotiated cipher suite's key exchange algorithm, and any negotiated extensions.

2.1. Certificate and CertificateVerify protection

If the server selects one of the cipher suites defined in this document, the client MUST symmetrically encrypt and integrity-protect the Certificate and the CertificateVerify messages.

The encryption and MAC algorithms are determined by the cipher_suite selected by the server and revealed in the ServerHello message.

The keys for the symmetric encryption and MAC are derived from the pre_master_secret.

This document reuses the hash algorithm and the two symmetric encryption modes defined by TLS: stream cipher encryption and block cipher encryption, in a manner dependent on the negotiated TLS version.

2.1.1. Stream cipher encryption

In stream cipher encryption, the client symmetrically encrypts the Certificate and the CertificateVerify messages without any padding byte. The encryption key cp_client_write_key is computed as described in Section 2.2.

The MAC notation slightly varies with the TLS version being employed. Symbolically, the MAC in this document is generated as follow:

In TLS version 1.2:

```
MAC(cp_client_write_MAC_key, plaintext)
```

The cp_client_write_MAC_key is generated as described in Section 2.2.

In TLS versions prior to 1.2:

```
HMAC_hash(cp_client_write_MAC_secret, plaintext)
```

The cp_client_write_MAC_secret is generated as described in Section 2.2.

Note that the MAC is computed before encryption. The stream cipher encrypts the entire block, including the MAC.

2.1.2. Block cipher encryption

In block cipher encryption, every block of plaintext encrypts to a block of ciphertext. All block cipher encryption is done in CBC (Cipher Block Chaining) mode, and all items that are block-ciphered will be an exact multiple of the cipher block length.

In block cipher encryption, the client uses an explicit initialization vector, generated as described through this document. The client adds a padding value to force the structure's length of each the Certificate and the CertificateVerify messages to be an integral multiple of the block cipher's block length, as it is described later through this document.

2.2. Key derivation

For all key exchange methods, the same algorithm is used to convert the pre_master_secret into the cp_key_block (credential protection key block). The cp_key_block MUST be deleted from memory as soon as possible during the TLS handshake, i.e.

- on the client: after encoding the CertificateVerify message;
- on the server: after decoding and verifying this message.

All the keys and parameters generated in this section are used only to encrypt and compute the MAC of the client Certificate and the CertificateVerify messages. The name of these keys includes the text "cp" to refer to this use.

The pending premaster secret is used as an entropy source. To generate the CP encryption and MAC keys, compute using the pending connection state (see Section 6.1 of [RFC5246])

```
cp_key_block = PRF(pre_master_secret, "cp key block",
                   SecurityParameters.server_random +
                   SecurityParameters.client_random);
```

until enough output has been generated. Then the cp_key_block is partitioned as follows:

Case TLS version 1.2:

```
cp_client_write_MAC_key[SecurityParameters.mac_key_length]
cp_client_write_key[SecurityParameters.enc_key_length]
```

Case TLS version 1.1:

cp_client_write_MAC_secret[SecurityParameters.hash_size]
cp_client_write_key[SecurityParameters.key_material_length]

Case TLS version 1.0:

cp_client_write_MAC_secret[SecurityParameters.hash_size]
cp_client_write_key[SecurityParameters.key_material_length]
cp_client_write_IV[SecurityParameters.IV_size]

Note 1: There are always four TLS connection states outstanding [RFC5246]: the current read and write states, and the pending read and write states. All records (conveying the Handshake messages) are processed under the current read and write states per standard TLS rules (i.e., usually no encryption or MAC will be used unless renegotiation is in progress).

When one of the ciphersuites described in this document is negotiated, the encryption and MAC keys generated above are used to encrypt the content of the Certificate and the CertificateVerify messages in the ciphersuite specific part of the TLS Handshake Layer, independent of the current processing in the TLS Record Layer.

Note 2: During the handshake, the client MUST send the Certificate message before the ClientKeyExchange message. Because the ClientKeyExchange message conveys the encrypted pre_master_secret,

- the client has to use the pre_master_secret before sending the ClientKeyExchange message in order to perform the credential protection key derivation necessary to encrypt the Certificate and the CertificateVerify messages;

- the server cannot decrypt and verify the content of the Certificate and the CertificateVerify messages until it has received the ClientKeyExchange message, which allows the server to assemble the pre_master_secret needed to perform the credential protection key derivation necessary to this end.

2.3. Structure of Certificate and CertificateVerify

The stream-ciphered, block-ciphered and digitally-signed structures vary with the TLS version being employed.

2.3.1. Certificate structure

2.3.1.1. Case TLS version 1.2

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

      struct {
         select (SecurityParameters.cipher_type) {
            case stream:
               stream-ciphered struct {
                  ASN.1Cert certificate_list<0..2^24-1>;
                  opaque MAC[SecurityParameters.mac_length];
               };

            case block:
               opaque IV[SecurityParameters.record_iv_length];
               block-ciphered struct {
                  ASN.1Cert certificate_list<0..2^24-1>;
                  opaque MAC[SecurityParameters.mac_length];
                  uint8 padding[Certificate.padding_length];
                  uint8 padding_length;
               };

         };
      } Certificate;
```

The MAC is generated as described in Section 2.1.1 (the plaintext is the certificate_list).

IV
: As part of the TLS Handshake, the standard TLS IV (Initialization Vector) is generated and therefore used by the TLS Record protocol. This document uses a second IV, generated in the same way as described in Section 6.2.3.2 of [RFC5246]. This IV is only used during the encryption/decryption of the content of the Certificate message (concatenation of certificate_list and MAC).

  The IV SHOULD be chosen at random, and MUST be unpredictable. For block ciphers, the IV length is SecurityParameters.record_iv_length which is equal to the SecurityParameters.block_size.

padding
: Padding that is added to force the length of the Certificate structure to be an integral multiple of the block cipher's block length. The padding MAY be any length up to 255 bytes, as long as it results in the length of the encrypted Certificate being an integral multiple of the block length. Lengths longer than

necessary might be desirable to frustrate attacks on a protocol that are based on analysis of the lengths of exchanged messages. Each uint8 in the padding data vector MUST be filled with the padding length value. The receiver MUST check this padding and SHOULD use the bad_record_mac alert to indicate padding errors.

padding_length
The padding length MUST be such that the total size of the Certificate structure is a multiple of the cipher's block length. Legal values range from zero to 255, inclusive. This length specifies the length of the padding field exclusive of the padding_length field itself.

2.3.1.2. Case TLS version 1.1

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

struct {
   select (SecurityParameters.cipher_type) {
      case stream:
         stream-ciphered struct {
            ASN.1Cert certificate_list<0..2^24-1>;
            opaque MAC[SecurityParameters.hash_size];
         };

      case block:
         block-ciphered struct {
            opaque IV[SecurityParameters.block_length];
            ASN.1Cert certificate_list<0..2^24-1>;
            opaque MAC[SecurityParameters.hash_size];
            uint8 padding[Certificate.padding_length];
            uint8 padding_length;
         };

   };
} Certificate;
```

The MAC is generated as described in Section 2.1.1 (the plaintext is the certificate_list). For the generation and handling of the IV see [RFC4346], Section 6.2.3.2; this document supports both sample algorithms described there. The padding and padding_length are generated and handled as described in Section 2.3.1.1.

2.3.1.3. Case TLS version 1.0

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

struct {
   select (SecurityParameters.cipher_type) {
      case stream:
         stream-ciphered struct {
            ASN.1Cert certificate_list<0..2^24-1>;
            opaque MAC[SecurityParameters.hash_size];
         };

      case block:
         block-ciphered struct {
            ASN.1Cert certificate_list<0..2^24-1>;
            opaque MAC[SecurityParameters.hash_size];
            uint8 padding[Certificate.padding_length];
            uint8 padding_length;
         };

   };
} Certificate;
```

The MAC is generated as described in Section 2.1.1 (the plaintext is the certificate_list).

The padding is generated as described in Section 2.3.1.1.

Note: With block ciphers in CBC mode (Cipher Block Chaining), the IV for the Certificate content is generated with the other keys and secrets, as described in Section 2.2. The IV for CertificateVerify content (Section 2.3.2.3) is the last ciphertext block from the Certificate content. For more details of TLS 1.0 IV handling, see Sections 6.1, 6.2.3.2, and 6.3, of [RFC2246].

2.3.2. CertificateVerify structure

2.3.2.1. Case TLS version 1.2

```
digitally-signed struct {
   opaque handshake_messages[handshake_messages_length];
} Signature;
```

The digitally-signed type is described in Sections 4.7 of [RFC5246]. We use the above shorthand type notation 'Signature' for the standard content of the CertificateVerify struct (Section 7.4.8 of [RFC4346])

in a similar manner as a variant of it was defined for TLS versions 1.1 and 1.0 (Section 7.4.3 in [RFC4346] and [RFC2246] -- see below).

```
   struct {
      select (SecurityParameters.cipher_type) {
         case stream:
            stream-ciphered struct {
               Signature signature;
               opaque MAC[SecurityParameters.mac_length];
            };

         case block:
            opaque IV[SecurityParameters.record_iv_length];
            block-ciphered struct {
               Signature signature;
               opaque MAC[SecurityParameters.mac_length];
               uint8 padding[CertificateVerify.padding_length];
               uint8 padding_length;
            };

      };
   } CertificateVerify;
```

The padding, IV and the MAC are described in Section 2.3.1.1, replacing Certificate with CertificateVerify and the certificate_list with the signature. Semantically, the CertificateVerify content is the signature and the MAC of the certificate_list. The basic Certificate Verify handshake message is described in Section 7.4.8 of [RFC5246].

2.3.2.2. Case TLS version 1.1

```
   struct {
      select (SecurityParameters.cipher_type) {
         case stream:
            stream-ciphered struct {
               Signature signature;
               opaque MAC[SecurityParameters.hash_size];
            };

         case block:
            block-ciphered struct {
               opaque IV[SecurityParameters.block_length];
               Signature signature;
               opaque MAC[SecurityParameters.hash_size];
               uint8 padding[CertificateVerify.padding_length];
               uint8 padding_length;
```

```
            };

         };
      } CertificateVerify;
```

The padding, IV and the MAC are described in Section 2.3.1.2, replacing Certificate with CertificateVerify and the certificate_list with the signature. Semantically, the CertificateVerify content is the signature and the MAC of the certificate_list. The Signature type and the basic Certificate Verify message structure for TLS version 1.1 are described in Sections 7.4.3 and 7.4.8 of [RFC4346].

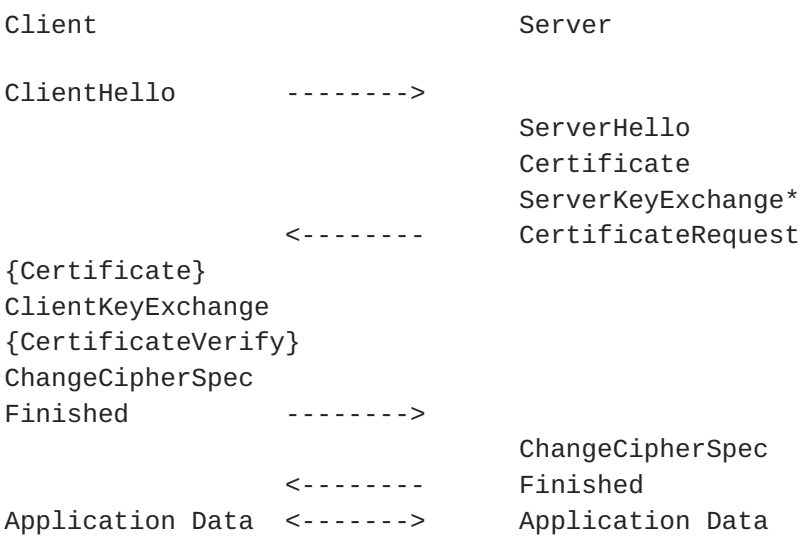
2.3.2.3. Case TLS version 1.0

```
      struct {
         select (SecurityParameters.cipher_type) {
            case stream:
               stream-ciphered struct {
                  Signature signature;
                  opaque MAC[SecurityParameters.hash_size];
               };

            case block:
               block-ciphered struct {
                  Signature signature;
                  opaque MAC[SecurityParameters.hash_size];
                  uint8 padding[CertificateVerify.padding_length];
                  uint8 padding_length;
               };

         };
      } CertificateVerify;
```

The Signature type and the basic CertificateVerify message structure for TLS version 1.0 are described in Sections 7.4.3 and 7.4.8 of [RFC2246].

With block ciphers in CBC mode, the IV is the last ciphertext block from the Certificate content. The padding and the MAC are generated as described in Section 2.3.1.3, replacing Certificate with CertificateVerify and the certificate_list with the signature.

2.4. Message Flow



* Indicates optional or situation-dependent messages that are not always sent.

{} Indicates messages with symmetrically encrypted and integrity-protected body.

For the DHE key exchange algorithm, the client always sends the ClientKeyExchange message conveying its ephemeral DH public key Yc.

For the ECDHE key exchange algorithm, the client always sends the ClientKeyExchange message conveying its ephemeral ECDH public key Yc.

Current TLS specifications note that if the client certificate already contains a suitable DH or ECDH public key, then Yc is implicit and does not need to be sent again and consequently, the client key exchange message will be sent, but it MUST be empty. Even if the client key exchange message is used to carry the Yc, using the same Yc will allow traceability. Consequently, static Diffie-Hellman SHOULD NOT be used with this document.

2.5. New ciphersuites

The cipher suites in Section 3 (CP_RSA Key Exchange Algorithm) use RSA based certificates to mutually authenticate an RSA exchange with client credential protection.

The cipher suites in Section 4 (CP_DHE Key Exchange Algorithm) use DHE_RSA or DHE_DSS to mutually authenticate an Ephemeral Diffie-Hellman (DHE) exchange.

The cipher suites in Section 5 (CP_ECDHE Key Exchange Algorithm) use ECDHE_RSA or ECDHE_ECDSA to mutually authenticate an Ephemeral Elliptic Curve Diffie-Hellman (ECDHE) exchange.

3. CP_RSA Key Exchange Algorithm

This section defines additional cipher suites that use RSA based certificates to authenticate an RSA exchange. These cipher suites give client credential protection.

CipherSuite	Key Exchange	Cipher	Hash
TLS_CP_RSA_WITH_RC4_128_MD5	RSA	RC4_128	MD5
TLS_CP_RSA_WITH_RC4_128_SHA	RSA	RC4_128	SHA1
TLS_CP_RSA_WITH_3DES_EDE_CBC_SHA	RSA	3DES_EDE	SHA1
TLS_CP_RSA_WITH_AES_128_CBC_SHA	RSA	AES_128_CBC	SHA1
TLS_CP_RSA_WITH_AES_256_CBC_SHA	RSA	AES_256_CBC	SHA1
TLS_CP_RSA_WITH_CAMELLIA_128_CBC_SHA	RSA	CAMELLIA_128_CBC	SHA1
TLS_CP_RSA_WITH_CAMELLIA_256_CBC_SHA	RSA	CAMELLIA_256_CBC	SHA1
TLS_CP_RSA_WITH_AES_128_CBC_SHA256	RSA	AES_128_CBC	SHA256
TLS_CP_RSA_WITH_AES_256_CBC_SHA384	RSA	AES_256_CBC	SHA384

4. CP_DHE Key Exchange Algorithm

This section defines additional cipher suites that use DHE as key exchange algorithm, with RSA or DSS based certificates to authenticate an Ephemeral Diffie-Hellman exchange. These cipher suites provide client credential protection and Perfect Forward Secrecy (PFS).

CipherSuite	Key Exchange	Cipher	Hash
TLS_CP_DHE_DSS_WITH_3DES_EDE_CBC_SHA	DHE_DSS	3DES_EDE_CBC	SHA1
TLS_CP_DHE_RSA_WITH_3DES_EDE_CBC_SHA	DHE_RSA	3DES_EDE_CBC	SHA1
TLS_CP_DHE_DSS_WITH_AES_128_CBC_SHA	DHE_DSS	AES_128_CBC	SHA1
TLS_CP_DHE_RSA_WITH_AES_128_CBC_SHA	DHE_RSA	AES_128_CBC	SHA1
TLS_CP_DHE_DSS_WITH_AES_256_CBC_SHA	DHE_DSS	AES_256_CBC	SHA1
TLS_CP_DHE_RSA_WITH_AES_256_CBC_SHA	DHE_RSA	AES_256_CBC	SHA1
TLS_CP_DHE_DSS_WITH_AES_128_CBC_SHA256	DHE_DSS	AES_128_CBC	SHA256
TLS_CP_DHE_RSA_WITH_AES_128_CBC_SHA256	DHE_RSA	AES_128_CBC	SHA256
TLS_CP_DHE_DSS_WITH_AES_256_CBC_SHA384	DHE_DSS	AES_256_CBC	SHA384
TLS_CP_DHE_RSA_WITH_AES_256_CBC_SHA384	DHE_RSA	AES_256_CBC	SHA384
TLS_CP_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA	DHE_DSS	CAMELLIA_128_CBC	SHA1
TLS_CP_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA	DHE_RSA	CAMELLIA_128_CBC	SHA1

TLS_CP_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA DHE_DSS CAMELLIA_256_CBC SHA1
TLS_CP_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA DHE_RSA CAMELLIA_256_CBC SHA1

5. CP_ECDHE Key Exchange Algorithm

This section defines additional cipher suites that use ECDHE as key exchange algorithm, with RSA or ECDSA based certificates to authenticate an Ephemeral ECDH exchange. These cipher suites provide client credential protection and PFS.

CipherSuite	Key Exchange	Cipher	Hash
TLS_CP_ECDHE_ECDSA_WITH_RC4_128_SHA	ECDHE_ECDSA	RC4_128	SHA1
TLS_CP_ECDHE_RSA_WITH_RC4_128_SHA	ECDHE_RSA	RC4_128	SHA1
TLS_CP_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA	ECDHE_ECDSA	3DES_EDE_CBC	SHA1
TLS_CP_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA	ECDHE_RSA	3DES_EDE_CBC	SHA1
TLS_CP_ECDHE_ECDSA_WITH_AES_128_CBC_SHA	ECDHE_ECDSA	AES_128_CBC	SHA1
TLS_CP_ECDHE_ECDSA_WITH_AES_256_CBC_SHA	ECDHE_RSA	AES_256_CBC	SHA1
TLS_CP_ECDHE_RSA_WITH_AES_128_CBC_SHA	ECDHE_RSA	AES_256_CBC	SHA1
TLS_CP_ECDHE_RSA_WITH_AES_256_CBC_SHA	ECDHE_RSA	AES_256_CBC	SHA1

6. Security Considerations

The security considerations described throughout [RFC2246], [RFC4346], [RFC5246], [RFC4347], and [RFC4492] apply here as well.

In order for the client to be protected against man-in-the-middle attacks, the client SHOULD verify that the server provided a valid certificate and that the received public key belongs to the server.

Because the question of whether this is the correct certificate is outside of TLS, applications that do implement credential protection cipher suites SHOULD enable the client to carefully examine the certificate presented by the server to determine if it meets its expectations. Particularly, the client MUST check its understanding of the server hostname against the server's identity as presented in the server Certificate message.

In the absence of an application profile specifying otherwise, the matching is performed according to the following rules, as described in [RFC4642]:

- The client MUST use the server hostname it used to open the connection (or the hostname specified in the TLS "server_name" extension [RFC4366]) as the value to compare against the server name as expressed in the server certificate. The client MUST NOT use any form of the server hostname derived from an

insecure remote source (e.g., insecure DNS lookup). CNAME canonicalization is not done.

- If a subjectAltName extension of type dNSName is present in the certificate, it MUST be used as the source of the server's identity.

- Matching is case-insensitive.

- A "*" wildcard character MAY be used as the left-most name component in the certificate. For example, *.example.com would match a.example.com, foo.example.com, etc., but would not match example.com.

- If the certificate contains multiple names (e.g., more than one dNSName field), then a match with any one of the fields is considered acceptable.

If the match fails, the client MUST either ask for explicit user confirmation or terminate the connection and indicate the server's identity is suspect.

Additionally, the client MUST verify the binding between the identity of the server to which it connects and the public key presented by this server. The client SHOULD implement the algorithm in Section 6 of [RFC5280] for general certificate validation, but MAY supplement that algorithm with other validation methods that achieve equivalent levels of verification (such as comparing the server certificate against a local store of already-verified certificates and identity bindings).

If the client has external information as to the expected identity of the server, the hostname check MAY be omitted.

It will depend on the application whether or not the server will have external knowledge of what the client's identity ought to be and what degree of assurance it needs to obtain of it. In any case, the server typically will have to check that the client has a valid certificate chained to an application-specific trust anchor it is configured with, following the rules of [RFC5280], before it successfully finishes the TLS handshake.

One widely accepted layering principle is to decouple service authorization from client authentication on access. We therefore recommend that authorization decisions be performed and communicated at the application layer after the TLS handshake has been completed.

Acknowledgment

People who should be acknowledged include Alfred Hoenes, Pasi Eronen and Eric Rescorla. Listing their names here does not mean that they endorse this document, but that they have reviewed it and have contributed to its improvement.

7. References

7.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2246] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999.

[RFC4132] Moriai, S., Kato, A., Kanda M., "Addition of Camellia Cipher Suites to Transport Layer Security (TLS)", RFC 4132, July 2005.

[RFC4346] Dierks, T. and E. Rescorla, "The TLS Protocol Version 1.1", RFC 4346, April 2005.

[RFC4347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security", RFC 4347, April 2006.

[RFC4366] Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J., and T. Wright, "Transport Layer Security (TLS) Extensions", RFC 4366, April 2006.

[RFC4492] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., Moeller, B., "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)", RFC 4492, May 2006.

[RFC4642] Murchison, K., Vinocur, J., Newman, C., "Using Transport Layer Security (TLS) with Network News Transfer Protocol (NNTP)", RFC 4642, October 2006.

[RFC5246] Dierks, T. and E. Rescorla, "The 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.

7.2. Informative References

[SSLTLS] Rescorla, E., "SSL and TLS: Designing and Building Secure Systems", Addison-Wesley, March 2001.

[CORELLA] Corella, F., "adding client identity protection to TLS", message on ietf-tls@lists.certicom.com mailing list, http://www.imc.org/ietf-tls/mail-archive/msg02004.html, August 2000.

[INTEROP] Pettersen, Y., "Clientside interoperability experiences for the SSL and TLS protocols", draft-ietf-tls-interoperability-00 (expired work in progress), October 2006.

[EAPIP] Urien, P. and M. Badra, "Identity Protection within EAP-TLS", draft-urien-badra-eap-tls-identity-protection-01.txt (expired work in progress), October 2006.

Author's Addresses

Ibrahim Hajjeh
INEOVATION
France

Email: hajjeh@ineovation.fr

Mohamad Badra
LIMOS Laboratory - UMR6158, CNRS
France

Email: mbadra@gmail.com