Credential Protection Ciphersuites for Transport Layer Security (TLS)
draft-hajjeh-tls-identity-protection-06.txt

Status of this Memo

By submitting this Internet-Draft, each author represents that any applicable patent or other IPR claims of which he or she is aware have been or will be disclosed, and any of which he or she becomes aware will be disclosed, in accordance with Section 6 of BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

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."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt
The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html

This Internet-Draft will expire on February 4, 2008.

Copyright Notice

Copyright (C) The IETF Trust (2008).

Abstract

This document defines a set of cipher suites to add client credential protection to the Transport Layer Security (TLS). 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
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 ................................................... 3
   1.1. Conventions used in this document ......................... 5
2. TLS credential protection overview ............................ 5
       2.1.1. Stream cipher encryption ............................ 6
       2.1.2. Block cipher encryption .............................. 6
   2.2. Key derivation ............................................ 7
   2.3. Structure of Certificate and CertificateVerify .......... 8
       2.3.1. Certificate structure ................................ 8
           2.3.1.1. Case TLS version 1.2 .......................... 8
           2.3.1.2. Case TLS version 1.1 .......................... 10
           2.3.1.3. Case TLS version 1.0 .......................... 11
       2.3.2. CertificateVerify structure ........................ 11
           2.3.2.1. Case TLS version 1.2 .......................... 11
           2.3.2.2. Case TLS version 1.1 .......................... 12
           2.3.2.3. Case TLS version 1.0 .......................... 13
   2.4. Message Flow ............................................. 14
   2.5. New ciphersuites ......................................... 14
3. CP_RSA Key Exchange Algorithm ................................ 15
4. CP_DHE Key Exchange Algorithm ................................ 15
5. CP_ECDHE Key Exchange Algorithm .............................. 16
6. Security Considerations ....................................... 16
7. IANA Considerations ......................................... 18
8. Acknowledgment ................................................. 19
9. References .................................................... 19
   9.1. Normative References ..................................... 19
   9.2. Informative References ................................... 20
Author’s Addresses ............................................. 20
Intellectual Property Statement ................................. 21
Disclaimer of Validity ......................................... 21
1. Introduction

The Transport Layer Security (TLS) protocol ([TLS1.2], [TLS1.1], [TLS1.0]) 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 change cipher spec message before the Certificate and the CertificateVerify messages and after the ClientKeyExchange message. The change cipher spec message is sent to notify the receiving party that subsequent messages will be protected under the CipherSpec and keys negotiated during the TLS Handshake. However, this solution requires a major change to the TLS machine state 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 [TLS1.2]: the client and the server establish a TLS
session with only server-side authentication and then bring up the
TLS session to establish a second TLS Handshake with mutual
authentication. This solution doesn’t require a change to TLS.
However, it requires more asymmetric cryptographic computations and
augments significantly the number of rounds trips. In fact,
renegotiation refers back to an asymmetric encryption/decryption and
to a full previously certificate chain verified public key, whose
chain was verified properly during the first handshake and stored in
the client session context. In addition, computation overhead
increases due to all second handshake messages encryption/decryption.
Regarding the round trips, their number increases dramatically
especially when small data packets are used to convey TLS messages.
Furthermore, the server is forced to complete a first TLS handshake
before it becomes able to confirm if the client has a certificate or
not.

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 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].

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, MD5, 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 server
hello 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 [TLS1.2], [TLS1.1], and [TLS1.0] standards 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 RFC-2119 [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 [TLS1.2], [TLSCTR], [TLSECC], [TLSAES], and [TLSCAM].

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.

<table>
<thead>
<tr>
<th>CipherSuite</th>
<th>Key Exchange</th>
<th>Cipher</th>
<th>Hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_CP_RSA_WITH_AES_128_CBC_SHA</td>
<td>RSA</td>
<td>AES_128_CBC</td>
<td>SHA1</td>
</tr>
<tr>
<td>TLS_CP_DHE_DSS_WITH_AES_128_CBC_SHA</td>
<td>DHE</td>
<td>AES_128_CBC</td>
<td>SHA1</td>
</tr>
</tbody>
</table>

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 encryption

If the server selects one of the cipher suites defined in this document, the client MUST symmetrically encrypt 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 server hello 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 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:

\[ \text{MAC}(cp\_client\_write\_MAC\_key, \text{plaintext}) \]

The \( cp\_client\_write\_MAC\_key \) is generated as described in section 2.2.

In TLS versions prior to 1.2:

\[ \text{HMAC}\_\text{hash}(cp\_client\_write\_MAC\_secret, \text{plaintext}) \]

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 premaster secret is used as an entropy source. To generate the encryption and MAC keys, compute

\[
\text{cp\_key\_block} = \text{PRF(} \text{pre\_master\_secret, } \text{"cp key block"}, \\
\text{SecurityParameters.server\_random + } \\
\text{SecurityParameters.client\_random)}; 
\]

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: 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 (CipherSpec.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 TLS Handshake, the 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 [TLS1.2]. This IV is only used during the encryption/decryption 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 of length SecurityParameters.record_iv_length which is equal to the SecurityParameters.block_size.

This document implements the same algorithms described in [TLS1.1] section 6.2.3.2 to generate the per-message IV (here the Certificate message):

1. Generate a cryptographically strong random string R of length CipherSpec.block_length. Place R in the IV field. Set the mask to R. Thus, the first cipher block will be encrypted as E(R XOR Data).

2. Generate a cryptographically strong random number R of length CipherSpec.block_length and prepend it to the plaintext prior to encryption. In this case either:

   a. The cipher may use a fixed mask such as zero.
   b. The CBC residue from the previous message may be used as the mask. This preserves maximum code compatibility with TLS 1.0 and SSL 3. It also has the advantage that it does not require the ability to quickly reset the IV, which is known to be a problem on some systems.

In either (2)(a) or (2)(b) the data (R || data) is fed into the encryption process. The first cipher block (containing E(mask XOR R)) is placed in the IV field. The first block of content contains E(IV XOR data).

mask

The actual value that the cipher XORs with the plaintext prior to encryption of the first cipher block of the Certificate content.
CBC residue
The last ciphertext block of the previous message.

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 (CipherSpec.cipher_type) {
        case stream:
            stream-ciphered struct {
                ASN.1Cert certificate_list<0..2^24-1>;
                opaque MAC[CipherSpec.hash_size];
            };
        case block:
            block-ciphered struct {
                opaque IV[CipherSpec.block_length];
                ASN.1Cert certificate_list<0..2^24-1>;
                opaque MAC[CipherSpec.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 and the IV 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 (CipherSpec.cipher_type) {
        case stream:
            stream-ciphered struct {
                ASN.1Cert certificate_list<0..2^24-1>;
                opaque MAC[CipherSpec.hash_size];
            };
        case block:
            block-ciphered struct {
                ASN.1Cert certificate_list<0..2^24-1>;
                opaque MAC[CipherSpec.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).

With block ciphers in CBC mode (Cipher Block Chaining) the initialization vector (IV) for the Certificate content is generated as described in section 2.2.

The padding is generated as described in section 2.3.1.1.

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 [TLS1.0].

2.3.2. CertificateVerify structure

2.3.2.1. Case TLS version 1.2

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

    struct {
        select (CipherSpec.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 generated as described in section 2.3.1.1, replacing Certificate with CertificateVerify and the certificate_list with the signature. The CertificateVerify content is the concatenation of the signature and the MAC. The digitally-signed type and the handshake_messages are described in [TLS1.2] section 7.4.8.

2.3.2.2. Case TLS version 1.1

struct {
  select (CipherSpec.cipher_type) {
    case stream:
      stream-ciphered struct {
        Signature signature;
opaque MAC[CipherSpec.hash_size];
      };
    case block:
      block-ciphered struct {
        opaque IV[CipherSpec.block_length];
        Signature signature;
opaque MAC[CipherSpec.hash_size];
        uint8 padding[CertificateVerify.padding_length];
        uint8 padding_length;
      };
  }
} CertificateVerify;

The padding, IV and the MAC are generated as described in section 2.3.1.2, replacing Certificate with CertificateVerify and the certificate_list with the signature. The CertificateVerify content is the concatenation of the signature and the MAC. The Signature type and structure are defined in [TLS1.1], sections 7.4.3 and 7.4.8.
2.3.2.3. Case TLS version 1.0

struct {
    select (CipherSpec.cipher_type) {
        case stream:
            stream-ciphered struct {
                Signature signature;
                opaque MAC[CipherSpec.hash_size];
            };
        case block:
            block-ciphered struct {
                Signature signature;
                opaque MAC[CipherSpec.hash_size];
                uint8 padding[CertificateVerify.padding_length];
                uint8 padding_length;
            };
    }
} CertificateVerify;

The Signature type and structure are defined in sections 7.4.3 and 7.4.8 of [TLS1.0].

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

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClientHello</td>
<td>ServerHello</td>
</tr>
<tr>
<td>Certificate</td>
<td>CertificateRequest</td>
</tr>
<tr>
<td>Certificate</td>
<td>{Certificate}</td>
</tr>
<tr>
<td>ClientKeyExchange*</td>
<td>CertificateRequest</td>
</tr>
<tr>
<td>{Certificate}</td>
<td></td>
</tr>
<tr>
<td>ClientKeyExchange</td>
<td></td>
</tr>
<tr>
<td>(CertificateVerify)</td>
<td></td>
</tr>
<tr>
<td>ChangeCipherSpec</td>
<td>ChangeCipherSpec</td>
</tr>
<tr>
<td>Finished</td>
<td>Finished</td>
</tr>
<tr>
<td>Application Data</td>
<td>Application Data</td>
</tr>
</tbody>
</table>

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

{} Indicates messages that are symmetrically encrypted.

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 a RSA exchange with the client credential protection.

The cipher suites in Section 4 (CP_DHE Key Exchange Algorithm) use DHE_RSA or DHE_DSS DSS to mutually authenticate an Ephemeral Diffie-Hellman (DHE) exchange.
The cipher suites in Section 5 (CP_ECDHE Key Exchange Algorithm) use ECDH_ECDSA or ECDHE_ECDSA to mutually authenticate an Ephemeral EC Diffie-Hellman (ECDHE) exchange.

3. CP_RSA Key Exchange Algorithm

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

<table>
<thead>
<tr>
<th>CipherSuite</th>
<th>Key</th>
<th>Cipher</th>
<th>Hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_CP_RSA_WITH_RC4_128_MD5</td>
<td>RSA</td>
<td>RC4_128</td>
<td>MD5</td>
</tr>
<tr>
<td>TLS_CP_RSA_WITH_RC4_128_SHA</td>
<td>RSA</td>
<td>RC4_128</td>
<td>SHA1</td>
</tr>
<tr>
<td>TLS_CP_RSA_WITH_3DES_EDE_CBC_SHA</td>
<td>RSA</td>
<td>3DES_EDE</td>
<td>SHA1</td>
</tr>
<tr>
<td>TLS_CP_RSA_WITH_AES_128_CBC_SHA</td>
<td>RSA</td>
<td>AES_128_CBC</td>
<td>SHA1</td>
</tr>
<tr>
<td>TLS_CP_RSA_WITH_AES_256_CBC_SHA</td>
<td>RSA</td>
<td>AES_256_CBC</td>
<td>SHA1</td>
</tr>
<tr>
<td>TLS_CP_RSA_WITH_AES_128_CTR_SHA</td>
<td>RSA</td>
<td>AES_128_CTR</td>
<td>SHA1</td>
</tr>
<tr>
<td>TLS_CP_RSA_WITH_AES_256_CTR_SHA</td>
<td>RSA</td>
<td>AES_256_CTR</td>
<td>SHA1</td>
</tr>
<tr>
<td>TLS_CP_RSA_WITH_CAMELLIA_128_CBC_SHA</td>
<td>RSA</td>
<td>CAMELLIA_128_CBC</td>
<td>SHA1</td>
</tr>
<tr>
<td>TLS_CP_RSA_WITH_CAMELLIA_256_CBC_SHA</td>
<td>RSA</td>
<td>CAMELLIA_256_CBC</td>
<td>SHA1</td>
</tr>
<tr>
<td>TLS_CP_RSA_WITH_AES_128_CBC_SHA256</td>
<td>RSA</td>
<td>AES_128_CBC</td>
<td>SHA256</td>
</tr>
<tr>
<td>TLS_CP_RSA_WITH_AES_256_CBC_SHA256</td>
<td>RSA</td>
<td>AES_256_CBC</td>
<td>SHA256</td>
</tr>
</tbody>
</table>

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 credentials protection and Perfect Forward Secrecy (PFS).

<table>
<thead>
<tr>
<th>CipherSuite</th>
<th>Key</th>
<th>Cipher</th>
<th>Hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_CP_DHE_DSS_WITH_3DES_EDE_CBC_SHA</td>
<td>DHE_DSS</td>
<td>3DES_EDE_CBC</td>
<td>SHA1</td>
</tr>
<tr>
<td>TLS_CP_DHE_RSA_WITH_3DES_EDE_CBC_SHA</td>
<td>DHE_RSA</td>
<td>3DES_EDE_CBC</td>
<td>SHA1</td>
</tr>
<tr>
<td>TLS_CP_DHE_RSA_WITH_AES_128_CBC_SHA</td>
<td>DHE_RSA</td>
<td>AES_128_CBC</td>
<td>SHA1</td>
</tr>
<tr>
<td>TLS_CP_DHE_RSA_WITH_AES_256_CBC_SHA</td>
<td>DHE_RSA</td>
<td>AES_256_CBC</td>
<td>SHA1</td>
</tr>
<tr>
<td>TLS_CP_DHE_RSA_WITH_AES_128_CTR_SHA</td>
<td>DHE_RSA</td>
<td>AES_128_CTR</td>
<td>SHA1</td>
</tr>
<tr>
<td>TLS_CP_DHE_RSA_WITH_AES_256_CTR_SHA</td>
<td>DHE_RSA</td>
<td>AES_256_CTR</td>
<td>SHA1</td>
</tr>
<tr>
<td>TLS_CP_DHE_RSA_WITH_AES_128_CBC_SHA256</td>
<td>DHE_RSA</td>
<td>AES_128_CBC</td>
<td>SHA256</td>
</tr>
<tr>
<td>TLS_CP_DHE_RSA_WITH_AES_256_CBC_SHA256</td>
<td>DHE_RSA</td>
<td>AES_256_CBC</td>
<td>SHA256</td>
</tr>
<tr>
<td>TLS_CP_DHE_RSA_WITH_AES_128_CTR_SHA256</td>
<td>DHE_RSA</td>
<td>AES_128_CTR</td>
<td>SHA256</td>
</tr>
<tr>
<td>TLS_CP_DHE_RSA_WITH_AES_256_CTR_SHA256</td>
<td>DHE_RSA</td>
<td>AES_256_CTR</td>
<td>SHA256</td>
</tr>
</tbody>
</table>
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 credentials protection and PFS.

<table>
<thead>
<tr>
<th>CipherSuite</th>
<th>Key Exchange Cipher</th>
<th>Hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_CP_ECDHE_ECDSA_WITH_RC4_128_SHA</td>
<td>ECDHE_ECDSA</td>
<td>RC4_128</td>
</tr>
<tr>
<td>TLS_CP_ECDHE_RSA_WITH_RC4_128_SHA</td>
<td>ECDHE_RSA</td>
<td>RC4_128</td>
</tr>
<tr>
<td>TLS_CP_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA</td>
<td>ECDHE_ECDSA</td>
<td>3DES_EDE_CBC</td>
</tr>
<tr>
<td>TLS_CP_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA</td>
<td>ECDHE_RSA</td>
<td>3DES_EDE_CBC</td>
</tr>
<tr>
<td>TLS_CP_ECDHE_ECDSA_WITH_AES_128_CBC_SHA</td>
<td>ECDHE_ECDSA</td>
<td>AES_128_CBC</td>
</tr>
<tr>
<td>TLS_CP_ECDHE_RSA_WITH_AES_128_CBC_SHA</td>
<td>ECDHE_RSA</td>
<td>AES_128_CBC</td>
</tr>
<tr>
<td>TLS_CP_ECDHE_ECDSA_WITH_AES_256_CBC_SHA</td>
<td>ECDHE_ECDSA</td>
<td>AES_256_CBC</td>
</tr>
<tr>
<td>TLS_CP_ECDHE_RSA_WITH_AES_256_CBC_SHA</td>
<td>ECDHE_RSA</td>
<td>AES_256_CBC</td>
</tr>
</tbody>
</table>

6. Security Considerations

The security considerations described throughout [TLS1.0], [TLS1.1], [TLS1.2], [DTLS], [TLSECC], and [TLSAES] apply here as well.

In order for the client to be protected against man-in-the-middle attack, 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 specification specifying otherwise, the matching is performed according to the following rules, as described in [RFC4642]:

Hajjeh & Badra          Expires February 2008                 [Page 16]
- The client MUST use the server hostname it used to open the connection (or the hostname specified in TLS "server_name" extension [TLSEXT]) 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 connect and the public key presented by this servers. The client SHOULD implement the algorithm in Section 6 of [PKICERT] 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.
7. IANA Considerations

This section provides guidance to the IANA regarding registration of values related to the credential protection cipher suites.

CipherSuite TLS_CP_RSA_WITH_RC4_128_MD5 = { 0XX,0XX };
CipherSuite TLS_CP_RSA_WITH_RC4_128_SHA = { 0XX,0XX };
CipherSuite TLS_CP_RSA_WITH_3DES_EDE_CBC_SHA = { 0XX,0XX };
CipherSuite TLS_CP_RSA_WITH_AES_128_CBC_SHA = { 0XX,0XX };
CipherSuite TLS_CP_RSA_WITH_AES_128_CTR_SHA = { 0XX,0XX };
CipherSuite TLS_CP_RSA_WITH_AES_256_CBC_SHA = { 0XX,0XX };
CipherSuite TLS_CP_RSA_WITH_AES_256_CTR_SHA = { 0XX,0XX };
CipherSuite TLS_CP_RSA_WITH_AES_128_CBC_SHA256 = { 00,0XX };
CipherSuite TLS_CP_RSA_WITH_AES_256_CBC_SHA256 = { 00,0XX };
CipherSuite TLS_CP_RSA_WITH_CAMELLIA_128_CBC_SHA = { 0XX,0XX };
CipherSuite TLS_CP_RSA_WITH_CAMELLIA_256_CBC_SHA = { 0XX,0XX };
CipherSuite TLS_CP_DHE_DSS_WITH_3DES_EDE_CBC_SHA = { 0XX,0XX };
CipherSuite TLS_CP_DHE_RSA_WITH_3DES_EDE_CBC_SHA = { 0XX,0XX };
CipherSuite TLS_CP_DHE_DSS_WITH_AES_128_CBC_SHA = { 0XX,0XX };
CipherSuite TLS_CP_DHE_RSA_WITH_AES_128_CBC_SHA = { 0XX,0XX };
CipherSuite TLS_CP_DHE_DSS_WITH_AES_256_CBC_SHA = { 0XX,0XX };
CipherSuite TLS_CP_DHE_RSA_WITH_AES_256_CBC_SHA = { 0XX,0XX };
CipherSuite TLS_CP_DHE_DSS_WITH_AES_128_CTR_SHA = { 0XX,0XX };
CipherSuite TLS_CP_DHE_RSA_WITH_AES_128_CTR_SHA = { 0XX,0XX };
CipherSuite TLS_CP_DHE_DSS_WITH_AES_256_CTR_SHA = { 0XX,0XX };
CipherSuite TLS_CP_DHE_RSA_WITH_AES_256_CTR_SHA = { 0XX,0XX };
CipherSuite TLS_CP_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA = { 0XX,0XX };
CipherSuite TLS_CP_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA = { 0XX,0XX };
CipherSuite TLS_CP_ECDHE_ECDSA_WITH_RC4_128_SHA = { 0XX,0XX };
CipherSuite TLS_CP_ECDHE_RSA_WITH_RC4_128_SHA = { 0XX,0XX };
CipherSuite TLS_CP_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA = { 0XX,0XX };
CipherSuite TLS_CP_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA = { 0XX,0XX };
CipherSuite TLS_CP_ECDHE_ECDSA_WITH_AES_128_CBC_SHA = { 0XX,0XX };
CipherSuite TLS_CP_ECDHE_RSA_WITH_AES_128_CBC_SHA = { 0XX,0XX };
CipherSuite TLS_CP_ECDHE_ECDSA_WITH_AES_256_CBC_SHA = { 0XX,0XX };
CipherSuite TLS_CP_ECDHE_RSA_WITH_AES_256_CBC_SHA = { 0XX,0XX };

Note: For implementation and deployment facilities, it is helpful to reserve a specific registry sub-range (minor, major) for credential protection cipher suites.
8. 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.

9. References

9.1. Normative References


9.2. Informative References


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


Author’s Addresses

Ibrahim Hajjeh
INOEVATION
France
Email: hajjeh@ineovation.com

Mohamad Badra
LIMOS Laboratory – UMR6158, CNRS
France
Email: badra@isima.fr
Intellectual Property Statement

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in BCP 78 and BCP 79.

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at http://www.ietf.org/ipr.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.

Disclaimer of Validity

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY, THE IETF TRUST AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Copyright Statement

Copyright (C) The IETF Trust (2008).

This document is subject to the rights, licenses and restrictions contained in BCP 78, and except as set forth therein, the authors retain all their rights.

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.