Appendix A. Protocol Data Structures and Constant Values

This section describes protocol types and constants.

A.1. Record Layer

struct { uint8 major; uint8 minor; } ProtocolVersion;

ProtocolVersion version = { 3, 3 }; /* TLS v1.2*/

enum { change_cipher_spec(20), alert(21), handshake(22), application_data(23), (255) } ContentType;

struct { ContentType type; ProtocolVersion version; uint16 length; opaque fragment[TLSPlaintext.length]; } TLSPlaintext;

struct { ContentType type; ProtocolVersion version; uint16 length; opaque fragment[TLSCompressed.length]; } TLSCompressed;

struct { ContentType type; ProtocolVersion version; uint16 length; select (SecurityParameters.cipher_type) { case stream: GenericStreamCipher; case block: GenericBlockCipher; case aead: GenericAEADCipher; } fragment; } TLSCiphertext;

stream-ciphered struct { opaque content[TLSCompressed.length]; opaque MAC[SecurityParameters.mac_length]; } GenericStreamCipher;

struct { opaque IV[SecurityParameters.record_iv_length]; block-ciphered struct { opaque content[TLSCompressed.length]; opaque MAC[SecurityParameters.mac_length]; uint8 padding[GenericBlockCipher.padding_length]; uint8 padding_length; }; } GenericBlockCipher;

struct { opaque nonce_explicit[SecurityParameters.record_iv_length]; aead-ciphered struct { opaque content[TLSCompressed.length]; }; } GenericAEADCipher;

A.2. Change Cipher Specs Message

struct { enum { change_cipher_spec(1), (255) } type; } ChangeCipherSpec;

A.3. Alert Messages

enum { warning(1), fatal(2), (255) } AlertLevel;

enum { close_notify(0), unexpected_message(10), bad_record_mac(20), decryption_failed_RESERVED(21), record_overflow(22), decompression_failure(30), handshake_failure(40), no_certificate_RESERVED(41), bad_certificate(42), unsupported_certificate(43), certificate_revoked(44), certificate_expired(45), certificate_unknown(46), illegal_parameter(47),

unknown_ca(48), access_denied(49), decode_error(50), decrypt_error(51), export_restriction_RESERVED(60), protocol_version(70),

```
   insufficient_security(71),
   internal_error(80),
   user_canceled(90),
   no_renegotiation(100),
   unsupported_extension(110),           /* new */
   (255)
```

} AlertDescription;

struct { AlertLevel level; AlertDescription description; } Alert;

A.4. Handshake Protocol

enum { hello_request(0), client_hello(1), server_hello(2), certificate(11), server_key_exchange (12), certificate_request(13), server_hello_done(14), certificate_verify(15), client_key_exchange(16), finished(20) (255) } HandshakeType;

struct { HandshakeType msg_type; uint24 length; select (HandshakeType) { case hello_request: HelloRequest; case client_hello: ClientHello; case server_hello: ServerHello; case certificate: Certificate; case server_key_exchange: ServerKeyExchange; case certificate_request: CertificateRequest; case server_hello_done: ServerHelloDone; case certificate_verify: CertificateVerify; case client_key_exchange: ClientKeyExchange; case finished: Finished; } body; } Handshake;

A.4.1. Hello Messages

struct { } HelloRequest;

struct { uint32 gmt_unix_time; opaque random_bytes[28]; } Random;

opaque SessionID<0..32>;

uint8 CipherSuite[2];

enum { null(0), (255) } CompressionMethod;

struct { ProtocolVersion client_version; Random random; SessionID session_id; CipherSuite cipher_suites<2..2^16-2>; CompressionMethod compression_methods<1..2^8-1>; select (extensions_present) { case false: struct {}; case true: Extension extensions<0..2^16-1>; }; } ClientHello;

struct { ProtocolVersion server_version; Random random; SessionID session_id; CipherSuite cipher_suite; CompressionMethod compression_method; select (extensions_present) { case false: struct {}; case true: Extension extensions<0..2^16-1>; }; } ServerHello;

struct { ExtensionType extension_type; opaque extension_data<0..2^16-1>; } Extension;

enum { signature_algorithms(13), (65535) } ExtensionType;

enum{ none(0), md5(1), sha1(2), sha224(3), sha256(4), sha384(5), sha512(6), (255) } HashAlgorithm; enum { anonymous(0), rsa(1), dsa(2), ecdsa(3), (255) } SignatureAlgorithm;

struct { HashAlgorithm hash; SignatureAlgorithm signature; } SignatureAndHashAlgorithm;

SignatureAndHashAlgorithm supported_signature_algorithms<2..2^16-1>;

A.4.2. Server Authentication and Key Exchange Messages

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

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

enum { dhe_dss, dhe_rsa, dh_anon, rsa,dh_dss, dh_rsa /* may be extended, e.g., for ECDH – see [TLSECC] */ } KeyExchangeAlgorithm;

struct { opaque dh_p<1..2^16-1>; opaque dh_g<1..2^16-1>; opaque dh_Ys<1..2^16-1>; } ServerDHParams; /* Ephemeral DH parameters */

struct { select (KeyExchangeAlgorithm) { case dh_anon: ServerDHParams params; case dhe_dss: case dhe_rsa: ServerDHParams params; digitally-signed struct { opaque client_random[32]; opaque server_random[32]; ServerDHParams params; } signed_params; case rsa: case dh_dss: case dh_rsa: struct {} ; /* message is omitted for rsa, dh_dss, and dh_rsa / / may be extended, e.g., for ECDH – see [TLSECC] */ } ServerKeyExchange;

enum { rsa_sign(1), dss_sign(2), rsa_fixed_dh(3), dss_fixed_dh(4), rsa_ephemeral_dh_RESERVED(5), dss_ephemeral_dh_RESERVED(6), fortezza_dms_RESERVED(20), (255) } ClientCertificateType;

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

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

struct { } ServerHelloDone;

A.4.3. Client Authentication and Key Exchange Messages

struct { select (KeyExchangeAlgorithm) { case rsa: EncryptedPreMasterSecret; case dhe_dss: case dhe_rsa: case dh_dss: case dh_rsa: case dh_anon: ClientDiffieHellmanPublic; } exchange_keys; } ClientKeyExchange;

struct { ProtocolVersion client_version; opaque random[46]; } PreMasterSecret;

struct { public-key-encrypted PreMasterSecret pre_master_secret; } EncryptedPreMasterSecret;

enum { implicit, explicit } PublicValueEncoding;

struct { select (PublicValueEncoding) { case implicit: struct {}; case explicit: opaque DH_Yc<1..2^16-1>; } dh_public; } ClientDiffieHellmanPublic;

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

A.4.4. Handshake Finalization Message

struct { opaque verify_data[verify_data_length]; } Finished;

A.5. The Cipher Suite

The following values define the cipher suite codes used in the ClientHello and ServerHello messages.

A cipher suite defines a cipher specification supported in TLS Version 1.2.

TLS_NULL_WITH_NULL_NULL is specified and is the initial state of a TLS connection during the first handshake on that channel, but MUST NOT be negotiated, as it provides no more protection than an unsecured connection.

```
CipherSuite TLS_NULL_WITH_NULL_NULL                = { 0x00,0x00 };
```

The following CipherSuite definitions require that the server provide an RSA certificate that can be used for key exchange. The server may request any signature-capable certificate in the certificate request message.

```
CipherSuite TLS_RSA_WITH_NULL_MD5                  = { 0x00,0x01 };
CipherSuite TLS_RSA_WITH_NULL_SHA                  = { 0x00,0x02 };
CipherSuite TLS_RSA_WITH_NULL_SHA256               = { 0x00,0x3B };
CipherSuite TLS_RSA_WITH_RC4_128_MD5               = { 0x00,0x04 };
CipherSuite TLS_RSA_WITH_RC4_128_SHA               = { 0x00,0x05 };
CipherSuite TLS_RSA_WITH_3DES_EDE_CBC_SHA          = { 0x00,0x0A };
CipherSuite TLS_RSA_WITH_AES_128_CBC_SHA           = { 0x00,0x2F };
CipherSuite TLS_RSA_WITH_AES_256_CBC_SHA           = { 0x00,0x35 };
CipherSuite TLS_RSA_WITH_AES_128_CBC_SHA256        = { 0x00,0x3C };
CipherSuite TLS_RSA_WITH_AES_256_CBC_SHA256        = { 0x00,0x3D };
```

The following cipher suite definitions are used for server- authenticated (and optionally client-authenticated) Diffie-Hellman. DH denotes cipher suites in which the server's certificate contains the Diffie-Hellman parameters signed by the certificate authority (CA). DHE denotes ephemeral Diffie-Hellman, where the Diffie-Hellman parameters are signed by a signature-capable certificate, which has been signed by the CA. The signing algorithm used by the server is specified after the DHE component of the CipherSuite name. The server can request any signature-capable certificate from the client for client authentication, or it may request a Diffie-Hellman certificate. Any Diffie-Hellman certificate provided by the client must use the parameters (group and generator) described by the server.

```
CipherSuite TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA       = { 0x00,0x0D };
CipherSuite TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA       = { 0x00,0x10 };
```

```
CipherSuite TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA     = { 0x00,0x13 };
CipherSuite TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA     = { 0x00,0x16 };
CipherSuite TLS_DH_DSS_WITH_AES_128_CBC_SHA       = { 0x00,0x30 };
CipherSuite TLS_DH_RSA_WITH_AES_128_CBC_SHA       = { 0x00,0x31 };
CipherSuite TLS_DHE_DSS_WITH_AES_128_CBC_SHA      = { 0x00,0x32 };
CipherSuite TLS_DHE_RSA_WITH_AES_128_CBC_SHA      = { 0x00,0x33 };
CipherSuite TLS_DH_DSS_WITH_AES_256_CBC_SHA       = { 0x00,0x36 };
CipherSuite TLS_DH_RSA_WITH_AES_256_CBC_SHA       = { 0x00,0x37 };
CipherSuite TLS_DHE_DSS_WITH_AES_256_CBC_SHA      = { 0x00,0x38 };
CipherSuite TLS_DHE_RSA_WITH_AES_256_CBC_SHA      = { 0x00,0x39 };
CipherSuite TLS_DH_DSS_WITH_AES_128_CBC_SHA256    = { 0x00,0x3E };
CipherSuite TLS_DH_RSA_WITH_AES_128_CBC_SHA256    = { 0x00,0x3F };
CipherSuite TLS_DHE_DSS_WITH_AES_128_CBC_SHA256   = { 0x00,0x40 };
CipherSuite TLS_DHE_RSA_WITH_AES_128_CBC_SHA256   = { 0x00,0x67 };
CipherSuite TLS_DH_DSS_WITH_AES_256_CBC_SHA256    = { 0x00,0x68 };
CipherSuite TLS_DH_RSA_WITH_AES_256_CBC_SHA256    = { 0x00,0x69 };
CipherSuite TLS_DHE_DSS_WITH_AES_256_CBC_SHA256   = { 0x00,0x6A };
CipherSuite TLS_DHE_RSA_WITH_AES_256_CBC_SHA256   = { 0x00,0x6B };
```

The following cipher suites are used for completely anonymous Diffie-Hellman communications in which neither party is authenticated. Note that this mode is vulnerable to man-in-the- middle attacks. Using this mode therefore is of limited use: These cipher suites MUST NOT be used by TLS 1.2 implementations unless the application layer has specifically requested to allow anonymous key exchange. (Anonymous key exchange may sometimes be acceptable, for example, to support opportunistic encryption when no set-up for authentication is in place, or when TLS is used as part of more complex security protocols that have other means to ensure authentication.)

```
CipherSuite TLS_DH_anon_WITH_RC4_128_MD5          = { 0x00,0x18 };
CipherSuite TLS_DH_anon_WITH_3DES_EDE_CBC_SHA     = { 0x00,0x1B };
CipherSuite TLS_DH_anon_WITH_AES_128_CBC_SHA      = { 0x00,0x34 };
CipherSuite TLS_DH_anon_WITH_AES_256_CBC_SHA      = { 0x00,0x3A };
CipherSuite TLS_DH_anon_WITH_AES_128_CBC_SHA256   = { 0x00,0x6C };
CipherSuite TLS_DH_anon_WITH_AES_256_CBC_SHA256   = { 0x00,0x6D };
```

Note that using non-anonymous key exchange without actually verifying the key exchange is essentially equivalent to anonymous key exchange, and the same precautions apply. While non-anonymous key exchange will generally involve a higher computational and communicational cost than anonymous key exchange, it may be in the interest of interoperability not to disable non-anonymous key exchange when the application layer is allowing anonymous key exchange.

New cipher suite values have been assigned by IANA as described in Section 12.

Note: The cipher suite values { 0x00, 0x1C } and { 0x00, 0x1D } are reserved to avoid collision with Fortezza-based cipher suites in SSL 3.

A.6. The Security Parameters

These security parameters are determined by the TLS Handshake Protocol and provided as parameters to the TLS record layer in order to initialize a connection state. SecurityParameters includes:

enum { null(0), (255) } CompressionMethod;

enum { server, client } ConnectionEnd;

enum { tls_prf_sha256 } PRFAlgorithm;

enum { null, rc4, 3des, aes } BulkCipherAlgorithm;

enum { stream, block, aead } CipherType;

enum { null, hmac_md5, hmac_sha1, hmac_sha256, hmac_sha384, hmac_sha512} MACAlgorithm;

/* Other values may be added to the algorithms specified in CompressionMethod, PRFAlgorithm, BulkCipherAlgorithm, and MACAlgorithm. */

struct { ConnectionEnd entity; PRFAlgorithm prf_algorithm; BulkCipherAlgorithm bulk_cipher_algorithm; CipherType cipher_type; uint8 enc_key_length; uint8 block_length; uint8 fixed_iv_length; uint8 record_iv_length; MACAlgorithm mac_algorithm; uint8 mac_length; uint8 mac_key_length; CompressionMethod compression_algorithm; opaque master_secret[48]; opaque client_random[32]; opaque server_random[32]; } SecurityParameters;

A.7. Changes to RFC 4492

RFC 4492 [TLSECC] adds Elliptic Curve cipher suites to TLS. This document changes some of the structures used in that document. This section details the required changes for implementors of both RFC 4492 and TLS 1.2. Implementors of TLS 1.2 who are not implementing RFC 4492 do not need to read this section.

This document adds a "signature_algorithm" field to the digitally- signed element in order to identify the signature and digest algorithms used to create a signature. This change applies to digital signatures formed using ECDSA as well, thus allowing ECDSA signatures to be used with digest algorithms other than SHA-1, provided such use is compatible with the certificate and any restrictions imposed by future revisions of [PKIX].

As described in Sections 7.4.2 and 7.4.6, the restrictions on the signature algorithms used to sign certificates are no longer tied to the cipher suite (when used by the server) or the ClientCertificateType (when used by the client). Thus, the restrictions on the algorithm used to sign certificates specified in Sections 2 and 3 of RFC 4492 are also relaxed. As in this document, the restrictions on the keys in the end-entity certificate remain.