6. The TLS Record Protocol

The TLS Record Protocol is a layered protocol. At each layer, messages may include fields for length, description, and content. The Record Protocol takes messages to be transmitted, fragments the data into manageable blocks, optionally compresses the data, applies a MAC, encrypts, and transmits the result. Received data is decrypted, verified, decompressed, reassembled, and then delivered to higher-level clients.

Four protocols that use the record protocol are described in this document: the handshake protocol, the alert protocol, the change cipher spec protocol, and the application data protocol. In order to allow extension of the TLS protocol, additional record content types can be supported by the record protocol. New record content type values are assigned by IANA in the TLS Content Type Registry as described in Section 12.

Implementations MUST NOT send record types not defined in this document unless negotiated by some extension. If a TLS implementation receives an unexpected record type, it MUST send an unexpected_message alert.

Any protocol designed for use over TLS must be carefully designed to deal with all possible attacks against it. As a practical matter, this means that the protocol designer must be aware of what security properties TLS does and does not provide and cannot safely rely on the latter.

Note in particular that type and length of a record are not protected by encryption. If this information is itself sensitive, application designers may wish to take steps (padding, cover traffic) to minimize information leakage.

6.1. Connection States

A TLS connection state is the operating environment of the TLS Record Protocol. It specifies a compression algorithm, an encryption algorithm, and a MAC algorithm. In addition, the parameters for these algorithms are known: the MAC key and the bulk encryption keys for the connection in both the read and the write directions. Logically, there are always four connection states outstanding: the current read and write states, and the pending read and write states. All records are processed under the current read and write states. The security parameters for the pending states can be set by the TLS Handshake Protocol, and the ChangeCipherSpec can selectively make either of the pending states current, in which case the appropriate current state is disposed of and replaced with the pending state; the pending state is then reinitialized to an empty state. It is illegal to make a state that has not been initialized with security parameters a current state. The initial current state always specifies that no encryption, compression, or MAC will be used.

The security parameters for a TLS Connection read and write state are set by

providing the following values:

connection end Whether this entity is considered the "client" or the "server" in this connection.

PRF algorithm An algorithm used to generate keys from the master secret (see Sections 5 and 6.3).

bulk encryption algorithm An algorithm to be used for bulk encryption. This specification includes the key size of this algorithm, whether it is a block, stream, or AEAD cipher, the block size of the cipher (if appropriate), and the lengths of explicit and implicit initialization vectors (or nonces).

MAC algorithm An algorithm to be used for message authentication. This specification includes the size of the value returned by the MAC algorithm.

compression algorithm An algorithm to be used for data compression. This specification must include all information the algorithm requires to do compression.

master secret A 48-byte secret shared between the two peers in the connection.

client random A 32-byte value provided by the client.

server random A 32-byte value provided by the server.

```
These parameters are defined in the presentation language as:


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;


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


/* The algorithms specified in CompressionMethod, PRFAlgorithm,
   BulkCipherAlgorithm, and MACAlgorithm may be added to. */
```

```
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;
```

The record layer will use the security parameters to generate the following six items (some of which are not required by all ciphers, and are thus empty):

```
client write MAC key
server write MAC key
client write encryption key
server write encryption key
client write IV
server write IV
```

The client write parameters are used by the server when receiving and processing records and vice versa. The algorithm used for generating these items from the security parameters is described in Section 6.3.

Once the security parameters have been set and the keys have been generated, the connection states can be instantiated by making them the current states. These current states MUST be updated for each record processed. Each connection state includes the following elements:

compression state The current state of the compression algorithm.

cipher state The current state of the encryption algorithm. This will consist of the scheduled key for that connection. For stream ciphers, this will also contain whatever state information is necessary to allow the stream to continue to encrypt or decrypt data.

MAC key The MAC key for this connection, as generated above.

sequence number Each connection state contains a sequence number, which is maintained separately for read and write states. The sequence number MUST be set to zero whenever a connection state is made the active state. Sequence numbers are of type uint64 and may not exceed 2^64-1. Sequence numbers do not wrap. If a TLS implementation would need to wrap a sequence number, it must renegotiate instead. A sequence number is incremented after each record: specifically, the first record transmitted under a particular connection state MUST use sequence number 0.

6.2. Record Layer

The TLS record layer receives uninterpreted data from higher layers in non-empty blocks of arbitrary size.

6.2.1. Fragmentation

The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^14 bytes or less. Client message boundaries are not preserved in the record layer (i.e., multiple client messages of the same ContentType MAY be coalesced into a single TLSPlaintext record, or a single message MAY be fragmented across several records).

```
struct {
    uint8 major;
    uint8 minor;
} ProtocolVersion;


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;
```

type The higher-level protocol used to process the enclosed fragment.

version The version of the protocol being employed. This document describes TLS Version 1.2, which uses the version { 3, 3 }. The version value 3.3 is historical, deriving from the use of {3, 1} for TLS 1.0. (See Appendix A.1.) Note that a client that supports multiple versions of TLS may not know what version will be employed before it receives the ServerHello. See Appendix E for discussion about what record layer version number should be employed for ClientHello.

length The length (in bytes) of the following TLSPlaintext.fragment. The length MUST NOT exceed 2^14.

fragment The application data. This data is transparent and treated as an independent block to be dealt with by the higher-level protocol specified by the type field.

Implementations MUST NOT send zero-length fragments of Handshake, Alert, or ChangeCipherSpec content types. Zero-length fragments of Application data MAY be sent as they are potentially useful as a traffic analysis countermeasure.

Note: Data of different TLS record layer content types MAY be interleaved. Application data is generally of lower precedence for transmission than other content types. However, records MUST be delivered to the network in the same order as they are protected by the record layer. Recipients MUST receive and process interleaved application layer traffic during handshakes subsequent to the first one on a connection.

6.2.2. Record Compression and Decompression

All records are compressed using the compression algorithm defined in the current session state. There is always an active compression algorithm; however, initially it is defined as CompressionMethod.null. The compression algorithm translates a TLSPlaintext structure into a TLSCompressed structure. Compression functions are initialized with default state information whenever a connection state is made active. [RFC3749] describes compression algorithms for TLS.

Compression must be lossless and may not increase the content length by more than 1024 bytes. If the decompression function encounters a TLSCompressed.fragment that would decompress to a length in excess of 2^14 bytes, it MUST report a fatal decompression failure error.

```
struct {
```

```
    ContentType type;       /* same as TLSPlaintext.type */
    ProtocolVersion version;/* same as TLSPlaintext.version */
    uint16 length;
    opaque fragment[TLSCompressed.length];
} TLSCompressed;
```

length The length (in bytes) of the following TLSCompressed.fragment. The length MUST NOT exceed 2^14 + 1024.

fragment The compressed form of TLSPlaintext.fragment.

```
Note: A CompressionMethod.null operation is an identity operation;
no fields are altered.


Implementation note: Decompression functions are responsible for
ensuring that messages cannot cause internal buffer overflows.
```

6.2.3. Record Payload Protection

```
The encryption and MAC functions translate a TLSCompressed
structure into a TLSCiphertext.  The decryption functions reverse
the process.  The MAC of the record also includes a sequence
number so that missing, extra, or repeated messages are
detectable.


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

type The type field is identical to TLSCompressed.type.

version The version field is identical to TLSCompressed.version.

length The length (in bytes) of the following TLSCiphertext.fragment. The length MUST NOT exceed 2^14 + 2048.

fragment The encrypted form of TLSCompressed.fragment, with the MAC.

6.2.3.1. Null or Standard Stream Cipher

Stream ciphers (including BulkCipherAlgorithm.null; see Appendix A.6) convert TLSCompressed.fragment structures to and from stream TLSCiphertext.fragment structures.

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

The MAC is generated as:

```
MAC(MAC_write_key, seq_num +
                      TLSCompressed.type +
                      TLSCompressed.version +
                      TLSCompressed.length +
                      TLSCompressed.fragment);
```

where "+" denotes concatenation.

seq_num The sequence number for this record.

MAC The MAC algorithm specified by SecurityParameters.mac_algorithm.

Note that the MAC is computed before encryption. The stream cipher encrypts the entire block, including the MAC. For stream ciphers that do not use a synchronization vector (such as RC4), the stream cipher state from the end of one record is simply used on the subsequent packet. If the cipher suite is TLS_NULL_WITH_NULL_NULL, encryption consists of the identity operation (i.e., the data is not encrypted, and the MAC size is zero, implying that no MAC is used). For both null and stream ciphers, TLSCiphertext.length is TLSCompressed.length plus SecurityParameters.mac_length.

6.2.3.2. CBC Block Cipher

For block ciphers (such as 3DES or AES), the encryption and MAC functions convert TLSCompressed.fragment structures to and from block TLSCiphertext.fragment structures.

```
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;
```

The MAC is generated as described in Section 6.2.3.1.

IV The Initialization Vector (IV) SHOULD be chosen at random, and MUST be unpredictable. Note that in versions of TLS prior to 1.1, there was no IV field, and the last ciphertext block of the previous record (the "CBC residue") was used as the IV. This was changed to prevent the attacks described in [CBCATT]. For block ciphers, the IV length is of length SecurityParameters.record_iv_length, which is equal to the SecurityParameters.block_size.

padding Padding that is added to force the length of the plaintext 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 TLSCiphertext.length 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 MUST use the bad_record_mac alert to indicate padding errors.

padding_length The padding length MUST be such that the total size of the GenericBlockCipher 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.

The encrypted data length (TLSCiphertext.length) is one more than the sum of SecurityParameters.block_length, TLSCompressed.length, SecurityParameters.mac_length, and padding_length.

Example: If the block length is 8 bytes, the content length (TLSCompressed.length) is 61 bytes, and the MAC length is 20 bytes, then the length before padding is 82 bytes (this does not include the

IV. Thus, the padding length modulo 8 must be equal to 6 in order to make the total length an even multiple of 8 bytes (the block length). The padding length can be 6, 14, 22, and so on, through 254. If the padding length were the minimum necessary, 6, the padding would be 6 bytes, each containing the value 6. Thus, the last 8 octets of the GenericBlockCipher before block encryption would be xx 06 06 06 06 06 06 06, where xx is the last octet of the MAC.

Note: With block ciphers in CBC mode (Cipher Block Chaining), it is critical that the entire plaintext of the record be known before any ciphertext is transmitted. Otherwise, it is possible for the attacker to mount the attack described in [CBCATT].

Implementation note: Canvel et al. [CBCTIME] have demonstrated a timing attack on CBC padding based on the time required to compute the MAC. In order to defend against this attack, implementations MUST ensure that record processing time is essentially the same whether or not the padding is correct. In general, the best way to do this is to compute the MAC even if the padding is incorrect, and only then reject the packet. For instance, if the pad appears to be incorrect, the implementation might assume a zero-length pad and then

compute the MAC. This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal.

6.2.3.3. AEAD Ciphers

For AEAD [AEAD] ciphers (such as [CCM] or [GCM]), the AEAD function converts TLSCompressed.fragment structures to and from AEAD TLSCiphertext.fragment structures.

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

AEAD ciphers take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [AEAD]. The key is either the client_write_key or the server_write_key. No MAC key is used.

Each AEAD cipher suite MUST specify how the nonce supplied to the AEAD operation is constructed, and what is the length of the GenericAEADCipher.nonce_explicit part. In many cases, it is

appropriate to use the partially implicit nonce technique described in Section 3.2.1 of [AEAD]; with record_iv_length being the length of the explicit part. In this case, the implicit part SHOULD be derived from key_block as client_write_iv and server_write_iv (as described in Section 6.3), and the explicit part is included in GenericAEAEDCipher.nonce_explicit.

The plaintext is the TLSCompressed.fragment.

The additional authenticated data, which we denote as additional_data, is defined as follows:

```
additional_data = seq_num + TLSCompressed.type +
                  TLSCompressed.version + TLSCompressed.length;
```

where "+" denotes concatenation.

The aead_output consists of the ciphertext output by the AEAD encryption operation. The length will generally be larger than TLSCompressed.length, but by an amount that varies with the AEAD cipher. Since the ciphers might incorporate padding, the amount of overhead could vary with different TLSCompressed.length values. Each AEAD cipher MUST NOT produce an expansion of greater than 1024 bytes. Symbolically,

```
AEADEncrypted = AEAD-Encrypt(write_key, nonce, plaintext,
```

```
                                        additional_data)
```

In order to decrypt and verify, the cipher takes as input the key, nonce, the "additional_data", and the AEADEncrypted value. The output is either the plaintext or an error indicating that the decryption failed. There is no separate integrity check. That is:

```
TLSCompressed.fragment = AEAD-Decrypt(write_key, nonce,
                                      AEADEncrypted,
                                      additional_data)
```

If the decryption fails, a fatal bad_record_mac alert MUST be generated.

6.3. Key Calculation

The Record Protocol requires an algorithm to generate keys required by the current connection state (see Appendix A.6) from the security parameters provided by the handshake protocol.

The master secret is expanded into a sequence of secure bytes, which is then split to a client write MAC key, a server write MAC key, a client write encryption key, and a server write encryption key. Each of these is generated from the byte sequence in that order. Unused values are empty. Some AEAD ciphers may additionally require a client write IV and a server write IV (see Section 6.2.3.3).

When keys and MAC keys are generated, the master secret is used as an entropy source.

To generate the key material, compute

```
key_block = PRF(SecurityParameters.master_secret,
                "key expansion",
                SecurityParameters.server_random +
                SecurityParameters.client_random);
```

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

```
client_write_MAC_key[SecurityParameters.mac_key_length]
server_write_MAC_key[SecurityParameters.mac_key_length]
client_write_key[SecurityParameters.enc_key_length]
server_write_key[SecurityParameters.enc_key_length]
client_write_IV[SecurityParameters.fixed_iv_length]
server_write_IV[SecurityParameters.fixed_iv_length]
```

Currently, the client_write_IV and server_write_IV are only generated for implicit nonce techniques as described in Section 3.2.1 of [AEAD].

Implementation note: The currently defined cipher suite which requires the most material is AES_256_CBC_SHA256. It requires 2 x 32 byte keys and 2 x 32

byte MAC keys, for a total 128 bytes of key material.