4. Presentation Language

This document deals with the formatting of data in an external representation. The following very basic and somewhat casually defined presentation syntax will be used. The syntax draws from several sources in its structure. Although it resembles the programming language "C" in its syntax and XDR [XDR] in both its syntax and intent, it would be risky to draw too many parallels. The purpose of this presentation language is to document TLS only; it has no general application beyond that particular goal.

4.1. Basic Block Size

The representation of all data items is explicitly specified. The basic data block size is one byte (i.e., 8 bits). Multiple byte data items are concatenations of bytes, from left to right, from top to bottom. From the byte stream, a multi-byte item (a numeric in the example) is formed (using C notation) by:

```
value = (byte[0] << 8*(n-1)) | (byte[1] << 8*(n-2)) |
        ... | byte[n-1];
```

This byte ordering for multi-byte values is the commonplace network byte order or big-endian format.

4.2. Miscellaneous

Comments begin with "*/* *and end with* */*".

Optional components are denoted by enclosing them in "[[ ]]" double brackets.

Single-byte entities containing uninterpreted data are of type opaque.

4.3. Vectors

A vector (single-dimensioned array) is a stream of homogeneous data elements. The size of the vector may be specified at documentation time or left unspecified until runtime. In either case, the length declares the number of bytes, not the number of elements, in the vector. The syntax for specifying a new type, T', that is a fixed- length vector of type T is

```
T T'[n];
```

Here, T' occupies n bytes in the data stream, where n is a multiple of the size of T. The length of the vector is not included in the encoded stream.

In the following example, Datum is defined to be three consecutive bytes that the protocol does not interpret, while Data is three consecutive Datum, consuming a total of nine bytes.

```
opaque Datum[3];      /* three uninterpreted bytes */
Datum Data[9];        /* 3 consecutive 3 byte vectors */
```

Variable-length vectors are defined by specifying a subrange of legal lengths, inclusively, using the notation <floor..ceiling>. When these are encoded, the actual length precedes the vector's contents in the byte stream. The length will be in the form of a number consuming as many bytes as required to hold the vector's specified maximum (ceiling) length. A variable-length vector with an actual length field of zero is referred to as an empty vector.

```
T T'<floor..ceiling>;
```

In the following example, mandatory is a vector that must contain between 300 and 400 bytes of type opaque. It can never be empty. The actual length field consumes two bytes, a uint16, which is sufficient to represent the value 400 (see Section 4.4). On the other hand, longer can represent up to 800 bytes of data, or 400 uint16 elements, and it may be empty. Its encoding will include a

two-byte actual length field prepended to the vector. The length of an encoded vector must be an even multiple of the length of a single element (for example, a 17-byte vector of uint16 would be illegal).

```
opaque mandatory<300..400>;
      /* length field is 2 bytes, cannot be empty */
uint16 longer<0..800>;
      /* zero to 400 16-bit unsigned integers */
```

4.4. Numbers

The basic numeric data type is an unsigned byte (uint8). All larger numeric data types are formed from fixed-length series of bytes concatenated as described in Section 4.1 and are also unsigned. The following numeric types are predefined.

```
uint8 uint16[2];
uint8 uint24[3];
uint8 uint32[4];
uint8 uint64[8];
```

All values, here and elsewhere in the specification, are stored in network byte (big-endian) order; the uint32 represented by the hex bytes 01 02 03 04 is equivalent to the decimal value 16909060.

Note that in some cases (e.g., DH parameters) it is necessary to represent integers as opaque vectors. In such cases, they are represented as unsigned integers (i.e., leading zero octets are not required even if the most significant bit is set).

4.5. Enumerateds

An additional sparse data type is available called enum. A field of type enum can only assume the values declared in the definition. Each definition is a different type. Only enumerateds of the same type may be assigned or compared. Every element of an enumerated must be assigned a value, as demonstrated in the following example. Since the elements of the enumerated are not ordered, they can be assigned any unique value, in any order.

```
enum { e1(v1), e2(v2), ... , en(vn) [[, (n)]] } Te;
```

An enumerated occupies as much space in the byte stream as would its maximal defined ordinal value. The following definition would cause one byte to be used to carry fields of type Color.

```
enum { red(3), blue(5), white(7) } Color;
```

One may optionally specify a value without its associated tag to force the width definition without defining a superfluous element.

In the following example, Taste will consume two bytes in the data stream but can only assume the values 1, 2, or 4.

```
enum { sweet(1), sour(2), bitter(4), (32000) } Taste;
```

The names of the elements of an enumeration are scoped within the defined type. In the first example, a fully qualified reference to the second element of the enumeration would be Color.blue. Such qualification is not required if the target of the assignment is well specified.

```
Color color = Color.blue;     /* overspecified, legal */
Color color = blue;           /* correct, type implicit */
```

For enumerateds that are never converted to external representation, the numerical information may be omitted.

```
enum { low, medium, high } Amount;
```

4.6. Constructed Types

Structure types may be constructed from primitive types for convenience. Each specification declares a new, unique type. The syntax for definition is much like that of C.

```
struct {
    T1 f1;
    T2 f2;
    ...
    Tn fn;
} [[T]];
```

The fields within a structure may be qualified using the type's name, with a syntax much like that available for enumerateds. For example, T.f2 refers to the second field of the previous declaration. Structure definitions may be embedded.

4.6.1. Variants

Defined structures may have variants based on some knowledge that is available within the environment. The selector must be an enumerated type that defines the possible variants the structure defines. There must be a case arm for every element of the enumeration declared in the select. Case arms have limited fall-through: if two case arms follow in immediate succession with no fields in between, then they

both contain the same fields. Thus, in the example below, "orange" and "banana" both contain V2. Note that this is a new piece of syntax in TLS 1.2.

The body of the variant structure may be given a label for reference. The mechanism by which the variant is selected at runtime is not prescribed by the presentation language.

```
struct {
    T1 f1;
    T2 f2;
    ....
    Tn fn;
     select (E) {
         case e1: Te1;
         case e2: Te2;
         case e3: case e4: Te3;
         ....
         case en: Ten;
     } [[fv]];
} [[Tv]];
```

For example:

```
enum { apple, orange, banana } VariantTag;


struct {
    uint16 number;
    opaque string<0..10>; /* variable length */
} V1;


struct {
    uint32 number;
    opaque string[10];    /* fixed length */
```

```
} V2;


struct {
    select (VariantTag) { /* value of selector is implicit */
        case apple:
          V1;   /* VariantBody, tag = apple */
        case orange:
        case banana:
          V2;   /* VariantBody, tag = orange or banana */
    } variant_body;       /* optional label on variant */
} VariantRecord;
```

4.7. Cryptographic Attributes

The five cryptographic operations – digital signing, stream cipher encryption, block cipher encryption, authenticated encryption with additional data (AEAD) encryption, and public key encryption – are designated digitally-signed, stream-ciphered, block-ciphered, aead- ciphered, and public-key-encrypted, respectively. A field's cryptographic processing is specified by prepending an appropriate key word designation before the field's type specification. Cryptographic keys are implied by the current session state (see Section 6.1).

A digitally-signed element is encoded as a struct DigitallySigned:

```
struct {
   SignatureAndHashAlgorithm algorithm;
   opaque signature<0..2^16-1>;
} DigitallySigned;
```

The algorithm field specifies the algorithm used (see Section 7.4.1.4.1 for the definition of this field). Note that the introduction of the algorithm field is a change from previous versions. The signature is a digital signature using those algorithms over the contents of the element. The contents themselves do not appear on the wire but are simply calculated. The length of the signature is specified by the signing algorithm and key.

In RSA signing, the opaque vector contains the signature generated using the RSASSA-PKCS1-v1_5 signature scheme defined in [PKCS1]. As discussed in [PKCS1], the DigestInfo MUST be DER-encoded [X680] [X690]. For hash algorithms without parameters (which includes SHA-1), the DigestInfo.AlgorithmIdentifier.parameters field MUST be NULL, but implementations MUST accept both without parameters and with NULL parameters. Note that earlier versions of TLS used a different RSA signature scheme that did not include a DigestInfo encoding.

In DSA, the 20 bytes of the SHA-1 hash are run directly through the Digital

Signing Algorithm with no additional hashing. This produces two values, r and s. The DSA signature is an opaque vector, as above, the contents of which are the DER encoding of:

```
Dss-Sig-Value ::= SEQUENCE {
    r INTEGER,
    s INTEGER
}
```

Note: In current terminology, DSA refers to the Digital Signature Algorithm and DSS refers to the NIST standard. In the original SSL and TLS specs, "DSS" was used universally. This document uses "DSA" to refer to the algorithm, "DSS" to refer to the standard, and it uses "DSS" in the code point definitions for historical continuity.

In stream cipher encryption, the plaintext is exclusive-ORed with an identical amount of output generated from a cryptographically secure keyed pseudorandom number generator.

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 AEAD encryption, the plaintext is simultaneously encrypted and integrity protected. The input may be of any length, and aead- ciphered output is generally larger than the input in order to accommodate the integrity check value.

In public key encryption, a public key algorithm is used to encrypt data in such a way that it can be decrypted only with the matching private key. A public-key-encrypted element is encoded as an opaque vector <0..2^16-1>, where the length is specified by the encryption algorithm and key.

RSA encryption is done using the RSAES-PKCS1-v1_5 encryption scheme defined in [PKCS1].

In the following example

```
stream-ciphered struct {
    uint8 field1;
    uint8 field2;
    digitally-signed opaque {
      uint8 field3<0..255>;
      uint8 field4;
    };
} UserType;
```

The contents of the inner struct (field3 and field4) are used as input for the signature/hash algorithm, and then the entire structure is encrypted with a stream cipher. The length of this structure, in bytes, would be equal to two bytes for field1 and field2, plus two bytes for the signature and hash algorithm,

plus two bytes for the length of the signature, plus the length of the output of the signing

algorithm. The length of the signature is known because the algorithm and key used for the signing are known prior to encoding or decoding this structure.

4.8. Constants

Typed constants can be defined for purposes of specification by declaring a symbol of the desired type and assigning values to it.

Under-specified types (opaque, variable-length vectors, and structures that contain opaque) cannot be assigned values. No fields of a multi-element structure or vector may be elided.

For example:

```
struct {
    uint8 f1;
    uint8 f2;
} Example1;


Example1 ex1 = {1, 4};  /* assigns f1 = 1, f2 = 4 */
```