Fragmenting TLS messages

TLS is certainly a big sprawling protocol. So unsurprisingly some of its loose ends aren't fully tied up. Take for example how at its lowest level it supports message transfer between server and client. The message layer (which implements the protocol) sits above a record layer, which in turn sits over TCP. TCP is a transport that takes raw data, packages it up and passes the result down via a data-link layer, to a physical layer from whence the bits are actually transmitted. The data-link layer ensures that all transmission errors are fixed, and the transport layer ensures that records are received in exactly the form and order that they were transmitted.

(Note that there is also DTLS which works over UDP, a different transport that does not deliver records in order and indeed may not deliver them at all. But that is another day's work).

So a TLS implementation takes protocol messages (which can be of virtually any length) and breaks them up into records for passing on to TCP. The absolute maximum plaintext record size is 16384 bytes. Records have a prepended header consisting of a one-byte content type, a two byte version number (always `0x0303`) and the two byte (16-bit) length of the payload, followed by the payload itself. Its notable what is not there, as the records are not numbered. Indeed there is no reason to number them, as thanks to TCP they always arrive in the right order.

The big question for an implementer is what size of record to use? Obviously the absolute maximum limit of 16384 bytes must be honoured (which means that fragmentation into records must be supported), but is something smaller more appropriate?

Before coming back to that lets consider another issue. After the initial clientHello almost everything is encrypted. This is done on a per record basis. So each record's payload is individually encrypted using an AEAD (Authenticated Encryption with Associated Data) cipher. This means that the whole record, header included, cannot be tampered with in transit, without such tampering being detected. To enable this, a 16-byte MAC is appended to the encrypted payload.

Now cryptographers hate their opponents to know anything about what they are transmitting. But you may have noticed that the content type is sent in the clear as part of the header. This arguably leaks information. Typical content types are Handshake Data (0x16) and Application Data (0x17). So a cunning ruse is used. The content type is always set to application data, and a final encrypted byte is appended to the payload which indicates its real type. So the record might appear as Application Data (0x17), but when properly decrypted it may be revealed to actually be Handshake Data (0x16). Clever.

To accommodate the encryption overhead, a ciphertext record has an absolute maximum size of 16384+256 bytes, the 256 to cover the extra 1 byte real record type, plus the 255 byte maximum AEAD overhead.

Which leaves us with one question. What size should we make fragments? Consider the issue from a client point of view, that is a client with memory constraints. Clearly it cannot process a record until it has buffered the whole record, at which time the MAC can be checked, the payload decrypted and the true type of the record revealed. So smaller records are better as less buffer space will be required. However each encrypted record incurs at least a 17 byte overhead, so larger records are more efficient. So there is a sweet spot somewhere between the two extremes.

To better understand what's going on, we need to realise that TCP does its own internal fragmentation as it strives to efficiently route our records to their destination. To make live easier for TCP its probably best to send it records that it hopefully may not have to fragment. And for TCP

the magic size, the MTU, is 1460 bytes. And therefore as we all love powers of 2, it would seem that 1024 byte records might be close to optimal.

Controlling fragment size

How does the TLS protocol handle this? Well a client can send a **`max_fragment_length`** extension to a server, asking for a maximum fragment size less that the absolute maximum, of 512, 1024, 2048 or 4096 bytes. It can ask, but the server can refuse to acknowledge it. Or it can acknowledge the request but nevertheless not act on it. Which is annoying. In only about 50% of cases does a web server acknowledge the request, and respond to it (the maximum agreed record size then applies to both parties).

Which isn't very satisfactory. But I guess a busy server has its own agenda which is to pump out messages as fast as it can, minimizing overhead, without worrying about wimpy clients. If a particular client has a problem with that, well its their problem. Tough.

It would be better if a client, and indeed a server, could demand a maximum limit, knowing that its demand would be respected. Hence the rather strange RFC8449 **`record_size_limit`** extension. Is it part of the TLS1.3 specification, or isn't it? Its not at all clear. The idea is simple. The client now demands a maximum record size (not less than 64 bytes) and the server will obey. The extension also works the other way around – a server can demand a maximum record size for records sent by the client.

Except when I tested it not a single server sent the extension, and not a single server responded to the extension when sent by the client. So this extension is not commonly supported. Clearly someone somewhere really doesn't want TLS to negotiate a maximum record size, and is quite happy to send most messages as single records, only fragmenting when they hit the absolute maximum.

However I think our client/server implementation should respect either one or other of these extensions. In the IoT space it is important to protect a constrained client from a server that wants to throw huge records at it.

Reconstructing messages

Now in the handshake phase some messages, like a large certificate chain, may be fragmented. And as each record is received the full chain must be reconstructed. So the message buffer may need to exceed the size of the record buffer. However in the application phase records may contain just lots of HTML which just needs to decrypted, displayed, and then immediately deleted.

Therefore we may conclude that the record buffer should ideally be of size 16384+256. In this case it would make sense to assume that all handshake messages are also less than that same limit (even though certificate chains could in theory be of any size). I suggest we simply reject any super-sized certificates.

In an IoT setting where both server and client are aware of and sympathetic to each other's constraints, we could use a much smaller record buffer size, and use one of above mechanisms to make sure all records are kept small enough. We can also insist on compact certificate chains.