Memory

How much memory does an application need? Many programmers go through their entire careers not caring about this, but in the constrained world of embedded devices it can matter a lot.

There are two main resources for memory, the stack and the heap. The heap is by far the most convenient, and for many applications it can be regarded as an inexhaustible resource. Just grab some memory whenever you need it (but don't forget to hand it back when you are finished with it).

Most of your computers memory is available to the heap, and in most contexts this will be many gigabytes in size. And even if you eat up all of that, the virtual memory system (a part of the operating system) kicks in and starts allocating memory from your hard disk. So it can certainly seem inexhaustible.

But what if you have only 32k bytes of RAM, you have no hard disk, and you have no operating system. Welcome to my world. Proceed on the assumption that you can always allocate from the heap, and your program will quickly crash. But the way in which it crashes will depend on the order in which functions are called, and hence may not be repeatable. A bug-ridden nightmare ensues.

So the alternative is the stack. Good news is that you never need to hand memory back, and you can re-use the same memory for many purposes, but the bad news is that you need to specify array sizes at compile time rather than at runtime. For a protocol like TLS this is a bit of a problem, as memory limits were never considered in its design. So implement TLS on a small embedded device and it will not be robust against memory exhaustion, either on the client or server side. And remember we are working in an environment where we might be under attack by a rogue client or a rogue server.

One of the many potential elephants in the room would be the certificates sent by the server to the client, which could be of any size. So the only alternative is to set maximum sizes for things. If these sizes are exceeded we will just have to drop the connection and the protocol will fail. However if we set the maximum sizes sensibly this should only very rarely happen.

Using only the stack for memory allocation can feel like programming with one hand tied behind your back. But it has benefits. We can work out exactly how much of the available memory we will require, and we can be sure that we will in all cases remain within that 32k (or whatever) limit.

Clearly this will be probably be more of an issue for the client than for the server, as a server usually (but not always) runs on a larger device. But the same approach can be used on both sides, with perhaps bigger limits on the server side. And a small client can to an extent defend itself from being swamped by offering only a very limited range of capabilities in its initial clientHello message, the first step in the protocol.

To this end maximum sizes for things have now been set in the tls1_3.h header file. These can of course be tweaked as we gain more experience.

What to put in and what to leave out?

It is clear that TLS1.3 is the output of a committee. Many of the features are dubious hang-overs from the past, some are gee-whiz ideas that will probably never be used – so why implement them?

For example a list of TLS1.3 implementations is available here

https://github.com/tlswg/tlswg-wiki/blob/master/IMPLEMENTATIONS.md

It is interesting to observe what is left in and what is left out of each of these implementations. Another useful site is

https://tls13.1d.pw/

which shows the handshake that takes place between your browser and a TLS1.3 compliant site. When using Chrome I observe the “padding” extension at the end of the browser’s clientHello. Refer to RFC7685 for a justification for this extension. Its there solely to deal with implementation bugs in certain TLS servers! Now that is plain silly. Simply refuse to connect to such servers until they fix the issue!

Here are some “features” which we may or may not decide to implement.

1. Client-side certificates for post-handshake client authentication

The whole thrust of TLS evolution has been to steer away from X.509 certificates and all of their painfulness. So introducing them for the client as well as the server seems like not a good idea. There are better ways for clients to authenticate (2-factor authentication for example).

2. Key Update

Either side can it seems demand at any time a key update. Now why would either party want to do that? Sounds like one of those gee-whizz ideas of dubious value which often introduce unexpected security holes. Still not sure about this one.

3. Record padding

This is quite different from the padding mentioned above. It is an anti-traffic-analysis option, which appears to me to be a good idea, but which appears to be not much used in practice.

4. 0-RTT

Zero return trips are touted as a big feature of TLS1.3. As is well known the TLS handshake is much more efficient with less over-and-backs between server and client. However 0-RTT makes security compromises, as with this feature we lose forward secrecy and are open to replay attacks. So maybe not for us.

It appears we need to go through all of the extensions in the TLS1.3 RFC and decide which are in and which are out, compliant with our design goals to (a) keep it secure and (b) keep it small.

More reading

Here is another interesting site (https://bearssl.org/), outlining the difficulties and challenges in implementing TLS.

Testing tool

It turns out that running our client against https://tls13.1d.pw is particularly useful, as this site randomly varies some parameters of its handshake, for example sometimes doing a HRR (Hello Retry Request), doing some padding, randomly asking for a Key Update etc.