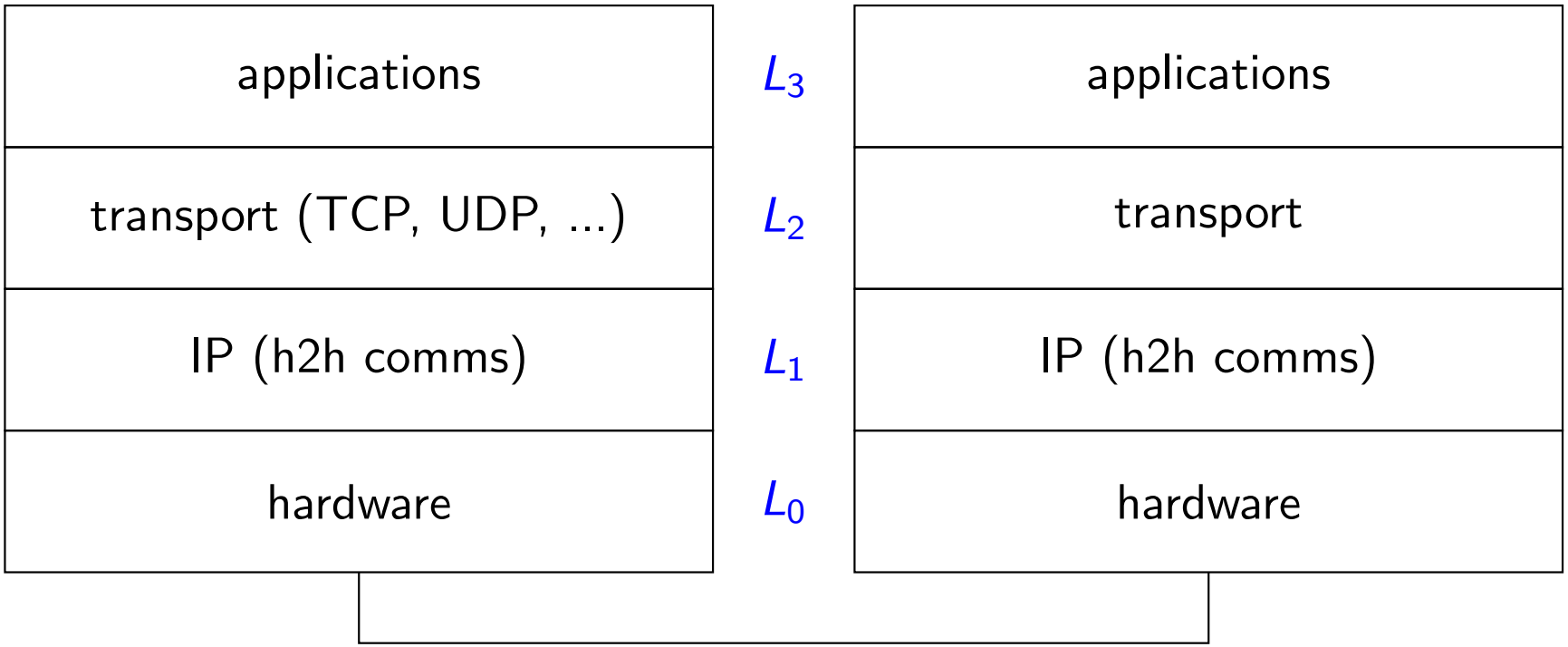
Securing the Transport Layer

Michael Brockway

March 5, 2018

Introduction



Applications that have to communicate data use transport layer services which use IP services, which use network hardware connected by data communications hardware.

The line connecting the two stacks is a *communications channel* using wire, wireless (two-way radio), fiber-optics or some combination.

Introduction - the problem

- In the basic setup, a third party could connect to the physical data communications channel and 'eavesdrop'.
- Alternatively, the third party can interpose themselves between the original parties (tampering with the physical coonections or wireless channels), intercept data packets and tamper with them...
- or destroy them...
- or inject data packets purporting to come from one of the original parties.
- Third parties can flood the channel with junk messages making it impossible for the original partis to function - (distributed) *denial of service* attack.

The applications need some defence against these.

DoS attacks are protected against by *firewalls* – processes that control ingress or egress of data packets from one network to another: these are often built-in to routers.

The other problems can be dealt with in a more general fashion ...

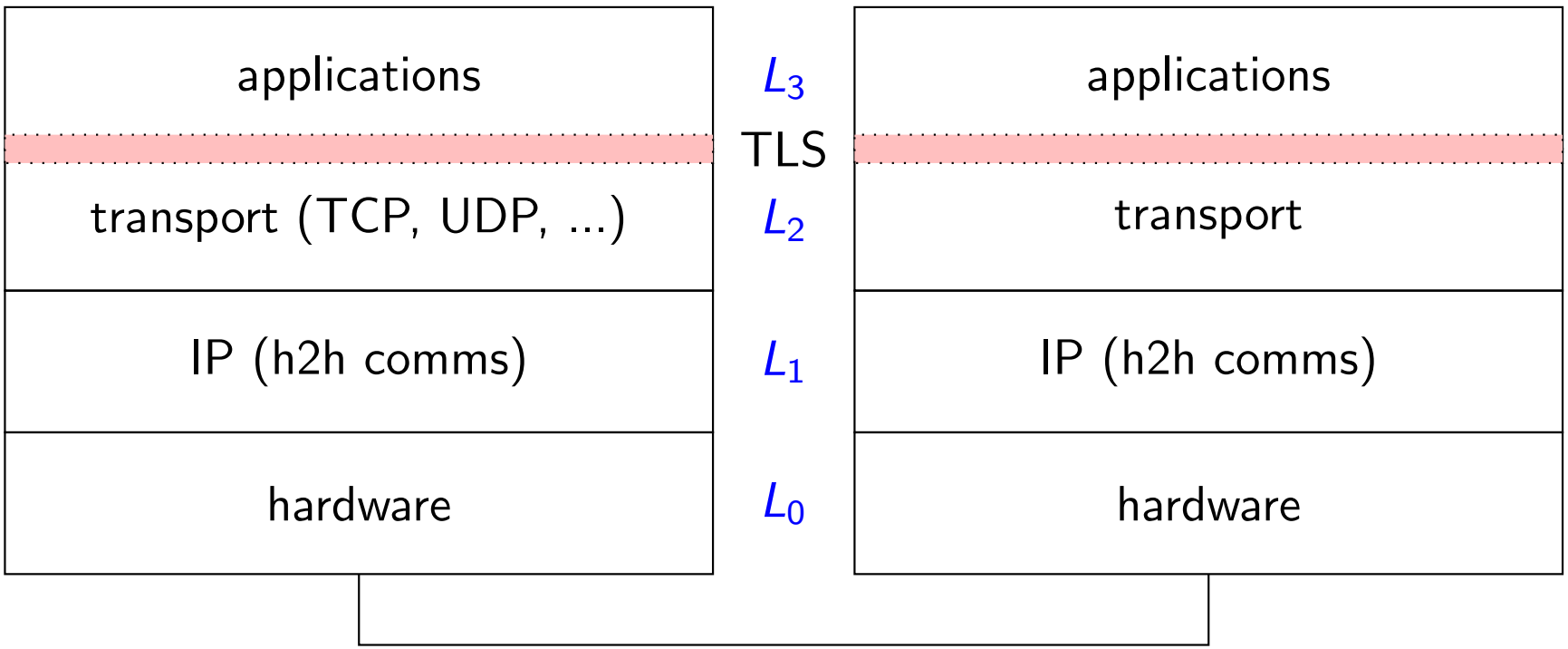
Transport layer security requirements

Fundamentally, applications communicate with one another,

- an application needs to know that a data packet cannot be read by unauthorised third parties - *privacy* or *confidentiality*.
- It needs to know it genuinely came from the network address given in the packet and was not injected by an imposter - *authenticity*.
- It needs to know the data contents are as sent and have not been tampered with - *integrity*.

The *assurances* are met by software sitting between the transport and application layers.

Transport layer security - basic setup



Transport-layer Security (TLS) is a 'thin' software layer sitting between the transport and application layers in the original TCP/IP stack.

Transport layer security - basic setup

TLS is designed to provide the required assurances:

- A data packet cannot be read by unauthorised third parties: *confidentiality* is assured use of a suitable *encryption* scheme. Data is encrypted before it is sent and decrypted when it received; only the legitimately communicating parties are able do this.
- A data packet comes with a *digital signature* which can be *authenticated* at the receiving end. Suitable digital signature schemes make it hard for a third party to fake a digital signature and thus assure *authenticity*.
- The *integrity* of a message is assured by using a *cryptographic hash function*. This function generates a *message digest* or *message authentication code* (MAC) which can be generated at the sending end and attached to the message; it is generated at the receiving end also, and the two values compared. Good hash functions make it difficult for a third party to alter the message while keeping the hash value the same.

Message and hash functions

A *hash function* takes as input a (possibly very long) sequence of bytes and returns a *hash value*, a number in a fixed range.

- For instance SHA256 returns a 256-bit number: in effect a number in the range $0 \ldots 2^{256} - 1$.

This is similar to the hash functions which support hash-tables of objects. What is special about a cryptographic hash function is that

- A small change in the input message (the byte sequence) always produces a big jump in the hash value;
- It is computationally infeasible to reverse-engineer hash values: to contrive a message whose hash is a given value;
- It is computationally infeasible to contrive two messages which give equal hash values.

A *collision-resistant* hash function is one with the first two of these properties. If it has all three properties it is *strongly collision-resistant*.

Message and hash functions

Thus, an attempt to tamper with message data will alter the hash. If a particular hash value is expected by the receiver, tampering will be detected.

- Contriving two messages that hash to the same value is a *birthday attack*: so called because the probability that in a group of n people, two will have a birthday on the same day is over 0.5 when $n > 23$. (See the wikipedia article on birthday attacks!)
- A strongly collision resistent hash function can thus be used to foil birthday attacks.

Usual TLS practice is

- The sender computes the hash of the message and appends the hash value to the message;
- the message+hash is encrypted and sent;
- the receiver decrypts the data and splits off the hash value;
- the receiver computes the hash of the received message and compares it with the received hash value.
- Tampering will inevitably have cause a mismatch.

Encryption

Encryption algorithms are in the public domain; their security lies in being *key-driven*

- The typical scheme is $c = E(k, m)$; $m = D(k', c)$;
- m is the *plain-text* message (byte sequence), c is the *cipher-text*.
- E is the *encryption* function which takes a *key* as well as the message.
- D is the companion *decryption* function which takes a *key* as well as the ciphertext.
- E, D are *inverses*: provided k' is the decryption key corresponding to encryption key k, then
- $\forall m : D(k', E(k, m)) = m$ and $\forall c : E(k, D(k', c)) = c$
- E, D are publicly known; but no-one can infer m from c without knowing k'.

Thus in TLS,

- the sender will send $E(k, m + h(m)) = c$ [m = message, $h(m)$ = hash]
- the receiver will receive c and decrypt: $D(k', c) = m' + h'$
- the receiver will check that $h(m') = h'$ to verify integrity.

Encryption

A *symmetric* encryption system is one with $k' = k$: the same key is used in encryption and decryption: it is shared between sender and receiver. Seems like a simplification but ...

- If a server, say, at PayPal, has to share pairs of keys with millions of customers, this is a massive management problem;
- How can it *agree* a shared key with each customer without an eaveropper discovering it?
- There has to be a different key with each customer (why?)

An *asymmetric* encryption system has $k' \neq k$:

- Now PayPal can publish its encryption key k_{PP}. PayPal's decription key k'_{PP} cannot be inferred; so anyone can send a message privately to PayPal and only PayPal can read it.
- PayPal's k_{PP} is its *public key*; k'_{PP} is its *private key*. Assymetric cryptosystems are more usually called *public key cryptosystems*.

Encryption

- PayPal customer Fred also has a public key k_F and a private key k'_F.
- A message $PP \rightarrow F$ is encrypted by PP with k_F (OK because k_F is public!) and recovered by F with k'_F;
- A message $F \rightarrow PP$ is encrypted by F with k_{PP} and recovered by PP with k'_{PP}.

We seem to have covered message integrity and confidentiality. What about authentication? Our public-key cryptographic system can be used for this too.

- PayPal can append its messages with its *digital signature* $sig = D(k'_{PP}, \text{“}PayPal\text{”})$.
- The hash(msg+sig) is computed and appended;
- The whole is encrypted: $E(k_F, (msg + sig + hash) = c$ and sent to Fred
- Fred decrypts: $D(k'_F, c)$ and recovers msg+sig+hash
- Fred extracts the hash and compares it to locally computed $hash(msg + sig)$ to check integrity.

Encryption

- To Fred, the message looks like plain text but the signature looks funny because if it is authentic, it is $D(k'_{PP}, \text{“}PayPal\text{”})$.
- Fred can check this by applying PP's public key: $E(k_{PP}, D(k'_{PP}, \text{“}PayPal\text{”}))$ - should get “PayPal” in plain text.
- This works because not only $\forall m : D(k', E(k, m)) = m$ (encryption followed by decryption) but also $\forall c : E(k, D(k', c)) = c$.

The public-key system is used with the roles of $E(k, -)$ and $D(k', -)$ reversed to do authentication.

Of course, all this work is done by PayPal's server and Fred's web browser and/or email client: not by an actual Fred.

Digital Certificates

How does Fred know the message from PayPal is really from PayPal and not from a 'man in the middle' masquerading as PayPal?

- PayPal's public key k_{PP} is registered with a trusted *certificate authority*.
- Fred's browser will verify k_{PP} with this authority before using it to check the authenticity of the message.

Re-enter symmetric cryptosystems

There is a problem with the scheme outlined so far. All current public key systems are very strong, but also slow: to slow for bulk encryption of megabytes of data in reasonable time.

Symmetric systems such as AES are 1000 or more times as fast. Unfortuately, the shared keys are impossible to manage on a large scale.

TLS therefore uses a hybrid system: a session between (say) PayPal and Fred

- start with an initial 'handshake' using a public key cryptosystem as above.
- Then a shared key is agreed for use in an agreed symmetric cryptosystem for the duration of the session.
- This negotion is kept private by use of the public-key system. There are also other 'key agreement protocols' such as Diffie-Hellman key agreement.
- From time to time during a long session, a new session key will be negotiated.

TLS practicalities

Examples of hash functions and cryptosystems actually used in transport-layer security will be outlined in the next two lectures. The last few slides here give a brief outline of the structure of TLS.

TLS originally developed from from SLL.

- the 'secure socket layer' originally developed in the 90s by Taher ElGamal, then chief scientist at Netscape.
- SSL went though a series of upgrades, through 2.0 to 3.0 as security loopholes were closed.
- SSL 3.0 was still vulnarable to a number of attacks - Heartbleed and POODLE, for instance: see RFCs 6176 and 7568.
- SSL used the RSA (Rivest-Shamir-Adleman) block cipher as its public-key system, but with static keys which undermine *forward security*. (The cipher itself is still considered secure.)
- SSL supported use the RC4 stream cipher which was eventually found to be vulnarable.
- The Data Encription Standard (DES, including 3DES) was still supported as the symmetric data encription system but by the early 2000s was easy to break.
- SSL used the MD5 and SHA1 hash algorithms which are considered too weak nowadays.

TLS practicalities

TLS 1.0 superceded SSL in 1999. See RFC 2246. The first big improvement was to prevent SSL logic from downgrading security for the sake of interoperability (POODLE exploited this).

TLS 1.1 is defined in RFC 4346 and included improved measures against *cipher block chaining* and *padding* attacks.

TLS 1.2 is defined in RFC 5246 (2008) and is current as of early 2018

- Hash algorithms MD5, SHA1 replaced by SHA256
- Advanced Encryption Standard (Rijndael) is the symmetric data encryption cipher of choice. Use in *counter mode* had vulnarabilities but *galois/counter* and *CCM* mitigate this.
- *authenticated encryption* combines AES encryption with integrity checking; some attacks exploited the separation of these operations.

TLS practicalities

As of early 2018, TLS 1.3 is in draft form. Proposals include

- ceasing to support MD5 and SHA-226 hash functions
- ceasing to support static RSA, static Diffie-Hellman key agreement
- disallowing RC4 or SSL negotiation for backwards compatibiltiy
- support for some new algorithms:
  - ChaCha20 stream cipher
  - Poly1305 MAC (hash function)
  - Ed25519 and Ed448 digital signature algorithms (they work in a similar fashion to PK digital signature checking)
  - x25519 and x448 key agreement protocols

SSH (RFC 4251)

'Secure SHell' - a protocol for secure log-in to a remote machine.

- Uses PK cryptography to authenticate the remote computer and allow it to authenticate the user.
- Supports tunneling, forwarding TCP ports
- Can transfer files using SSH file transfer (SFTP) or secure copy (SCP) protocols.

Cient-server model.

- On Unix-type systems 'ssh' (at the command prompt) starts a SSH client: command is

  ```
  $ ssh user@hostURL
  ```

- On Windows systems 'PuTTY'

SSH - uses

- login to a shell on a remote host: replaces Telnet, rlogin
- Canset up passwordless login to a remote server, eg using OpenSSH
- Secure file transfer
- forwarding or tunneling a port
- use as a full-fledged encrypted VPN (OpenSSH only)
- securely mounting a directory on a remote server as a filesystem on a local computer
- automated remote monitoring and management of servers using these mechanisms.

HTTP over TLS (RFC 2818)

HTTP, the communications protocol used by web browsers and servers, adapted for secure communication:

- The protocol messages are secured (encrypted, authenticated, integrity-checked) by TLS.
- You know your browser is using this when you seen 'HTTPS'
- Authenticates the accessed website and assures privacy and integrity of exchanged data.
- Protects against 'man-in-the-middle' attacks.
- Gives reasonable assurance that one is communicating, without interference by attackers, with the intended website, not an impostor.
- Widely used by banks, payment pages of e-commerce sites; there arre obvious applications everywhere a web-based service handles sensative data.

Further reading

RFCs

- RFC 6176 https://tools.ietf.org/html/rfc6176
- RFC 7568 https://tools.ietf.org/html/rfc7568 (SSL vulnarabilities)
- RFC 2246 https://tools.ietf.org/html/rfc2246 (TLS 1.0)
- RFC 4346 https://tools.ietf.org/html/rfc2246 (TLS 1.1)
- RFC 5246 https://tools.ietf.org/html/rfc5246 (TLS 1.2)
- RFC 4251 https://tools.ietf.org/html/rfc5246 (SSH)
- RFC 2818 https://tools.ietf.org/html/rfc5246 (https)