Ch.4: Transport Layer Security (TLS)

Ciro S. Costa

Jul 07, 2015

Because of the layered architecture of network protocols, running an application over TLS is no different from communicating directly over TCP. There are minimal application modifications to be done to make the deliver over TLS.

TLS

TLS (Transport Layer Security) was implemented at the application layer, directly on top of TCP - a reliable transport. By doing this it enabled protocols above it (http, email, im, etc) to operate unchanged while providing communication security. Nowadays it has been adapted to run over datagram protocols such as UDP (see *Datagram Transport Layer Secutiry* - DTLS - protocol) as well.

It aims to provide:

- **encryption**: for obfuscation
- **authentication**: for verifying the validity of provided identification material
- **data integrity**: detect tampering and forgery

These are guaranteed by establishing a cryptographically secure data channel with the peers agreeing on which ciphersuites will be used and the keys used to encrypt/decrypt the data. They also have to agree on the message framing mechanism which is responsible for signing each message with a *message authentication code* (`MAC`, kind of a checksum) to ensure both integrity and authenticity.

Public Key (asymmetric) Cryptography is a class of cryptographic protocols based on algorithms that require two separate keys, one of which is secret and one of which is public. Although different, these two parts are somehow matematically linked (there's also no possibility of guessing one from another). It differs from the symmetric one as there's no need for

two parties relying on the same key (that must be shared somehow in a safe way - but how if we have not yet established a safe way of doing?).

With public key crypto we generate two keys: one that will encrypt, another that will decrypt. Pick one of them and establish it as public (publish it however you want). Hold the other as private. Here comes some examples:

- Usage example 1: Person A generates pair (publicA, privateA). Person B generates (publicB, privateB). Each one knows each other's public key. As person A wants to talk privatly with B, it uses publicB to encrypt its message so that when B receives it it can decrypt (because he'll have privateB). Ok, the message it perfectly encrypted but person B doesn't really know if person A was the actual guy who send him a message.
- Usage example 2: Same situation as above but now person A, before sending a message to person B, encrypts the message with its own private key (privateA). The message is now out there and anyone can decrypt (as its public key - publicA - is on the internet). Even though the content of the message is not a surprise to anyone anymore, it's guaranteed that personA whas the one that wrote that (as only personA will have the privateA to encrypt it in the first place).
- Usage example 3: combine both of the examples and you have authenticity and privacy guaranteed.

Note that public key cryptography is slower that symmetric key. For this reason the TLS setup will run asymmetric crypto only on the first phase (when exchanging the symmetric key - during session setup of the TLS tunnel).

More than the normal SYN, SYN ACK, ACK flow from TCP we must add another layer of handshakes.

Handshakes

1. TCP Handshake
2. client sends specs in plain text such as TLS version, ciphersuites, etc
3. server picks tls version, decides other things, attaches its certificate and sends that info back to the client
4. if client is happy with the certificate and others, generates a symmetric key, encrypts it with the server's public key and tells the server to switch to encrypted communication. 4.1 From know on, everything is encrypted
5. the server decrypts the symmetric key sent by the clients, checks the integrity (through MAC) and returns an encrypted "finished" message back to the client

6. The client decrypts the message with the symmetric key generated earlier, verifies the MAC and if everything is OK then the tunnel is established and ready to run.

Notice that more RTT (two at least) are required.

New protocols through data obfuscation

Because TLS obfuscates data we can deliver any kind of protocols on the web without needing to worry about proxies and other intermediaries as these parties can't know about the data that is passing through them. Another great feature that TLS brings is the fact that we can rely on the 443 port and don't need to configure all clients to search for another port.

In the protocol negotiation side TLS helps with one more thing: ALPN (*Application Layer Protocol Negotiation*). This lets us specify the protocol during the TLS handshake phase (when we had already established a reliable transport throguh TCP - the client specifies which protocols it supports and the server then selects and confirms the protocol; or drops the connection). Note that we could use the new HTTP2.0 Upgrade mechanism in a non-secure channel but that would consume the same amount of RTT as in the TLS way without Upgrade.

Multiple Certificates

Using an extension (SNI - *Server Name Indication*) it's possible to use multiple certificates (for multiple domains) for the same IP and Port by letting the client indicate which hostname it is attempting to connect to at the start of the handshaking process (then letting the server decide which certificate to use). It is the equivalent of Virtual Hosting but for HTTPS.

Virtual Hosting is a method for hosting multiple domain names, with separate handling of each name, on a single server (or pool of servers). This allows one server to share its resources, such as memory and processor cycles, without requiring all services provided to use the same host name.

Session Resumption

After the first TLS handshake if fully made there's a way of caching the results of the first phase of the TLS handshake (in which infos about the ciphersuite and certificate is passed). The server maintains a cache of session Ds and the parameters for each peer. The client also stores the session ID and then

includes this ID in its subsequent messages in the TLS handshakes (which is then abbreviated by one RTT).

ps: as more clients attempt to send a burst of requests all in parallel, it's common to have an intentional way for the first TLS connection to complete before openning all of the other connections in parallel (so that it reuses the parameters for all of them).

The server-side cache nowadays was removed and replaced by *session caching* and *stateless resumption* mechanism through *Session Tickets*, which is a way of the TLS server encapsulating the session state into a ticket and forwarding it to the client (using it then to resume a session).

Chain of Trust

The idea of a chain of trust is about establishing a way of determining whether someone who i trust trusts in another party which i don't know (but my friend knows). In the public key analogy my friend knows the other if my friend signs the other guy's messages with its public key. This is the idea of transitive trust. To determine the browser's chain of trust there are 3 ways:

- Manually Specified Certificates
- Certificate Authorities: a CA is a trusted third party that is trusted by both the subject (owner) of the certificate and the party relying upon the certificate
- The Browser and OS: predefined list of trusted CAs

Checking a certificate status if just a matter of querying OCSP (*Online Certificate Status Protocol*) or checking the CLR (*Certificate Revocation List*). Maintaining the list and an endpoint for OCSP (realtime) is a CA's duty. It must ensure that the service is up and globally available at all times. Because the client must block on OCSP requests before proceeding with the navigation, this might affect the latency.

Optimizing

As TLS runs over TCP, optmize for TCP. Also, special attention to early termination (placing servers closer to the user). Don't just optimize in this way for delivery of static assets, optimize it also for TLS session termination. Remember that two ways have to be encrypted: CDN to Clients and Server to CDN.

There also exists TLS compression but one should disable it. It will recompress already compressed data as well as exposing to unsecure activitives.

Verifying the chain of trust is also a time-consuming process. We MUST provide all the intermediate certificates in the chain so that the browser will not be forced to pause the verification and fetch the intermediate certificate on their own, verify it and then continue (which would require a new DNS lookup, TCP connection and HTTP GET request). It's also important to not include unnecessary certificate in the chain.

OCSP might be optimized as well by including the response of it so that the browser does not have to go search it. A problem that might arise from doing this is overflowing the TCP congestion window.

HTTP Strict Transport Security (HSTS)

> HSTS is a security policy mechanism that allows the server to declare access rules to a compliant browser via a simple HTTP header, instructing the user-agent to enforce rules like making all of the requests to the origin via HTTPS, converting insecure links and client requests to HTTPS, displaying error messages which can't be circumvented in case of certificate error and definition of lifetime of the HSTS ruleset. It's also necessary to protect secure HTTPS websites against downgrade attacks.