TLS 1.3

Eric Rescorla

Mozilla

ekr@rtfm.com

Overview

- Background/Review of TLS
- Some problems with TLS 1.2
- Objectives for TLS 1.3
- What does TLS 1.3 look like?
- Open issues/schedule/etc.

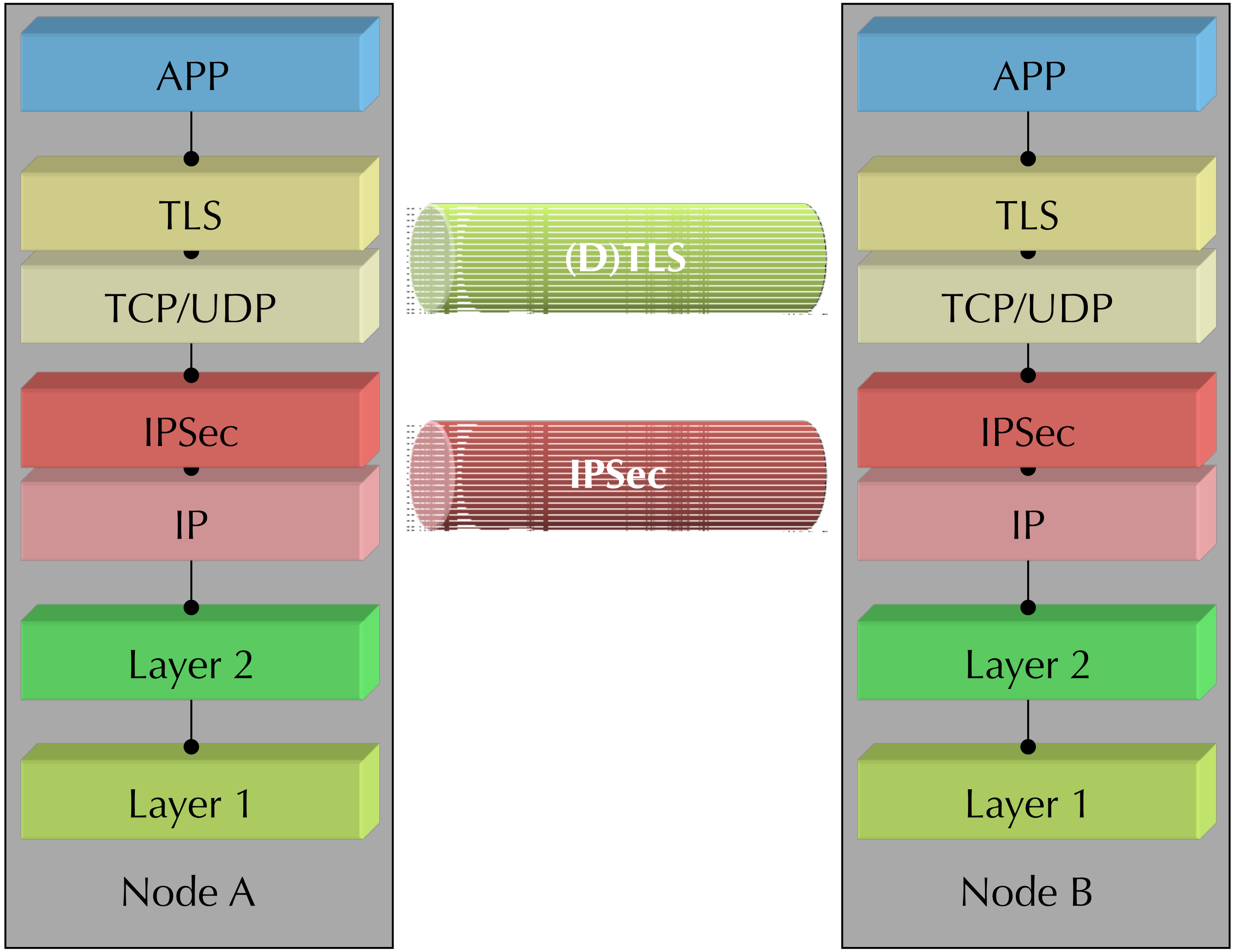
What is Transport Layer Security?

- Probably the Internet's most important security protocol
- Designed over 20 years ago by Netscape for Web transactions
  - Back then, called Secure Sockets Layer
- But used for just about everything you can think of
  - HTTP
  - SSL-VPNs
  - E-mail
  - Voice/video
  - IoT
- Maintained by the Internet Engineering Task Force
  - We're now at version 1.2

A Secure Channel

- Client connects to a known server (e.g., it has the domain name)
- Server is (almost) always authenticated by TLS
- Client may or may not be authenticated by TLS
  - Often authenticated by the application, e.g., with a password
- After setup, data is encrypted and authenticated
  - Though what "authenticated" means to the server is fuzzy

IPSec and Transport Layer Security (TLS)
Security at layers 3 and 4



- Security services
  - Authentication
  - Integrity protection
  - Confidentiality
- Security services become independent, wherever possible, from upper and lower layers

TLS: introduction

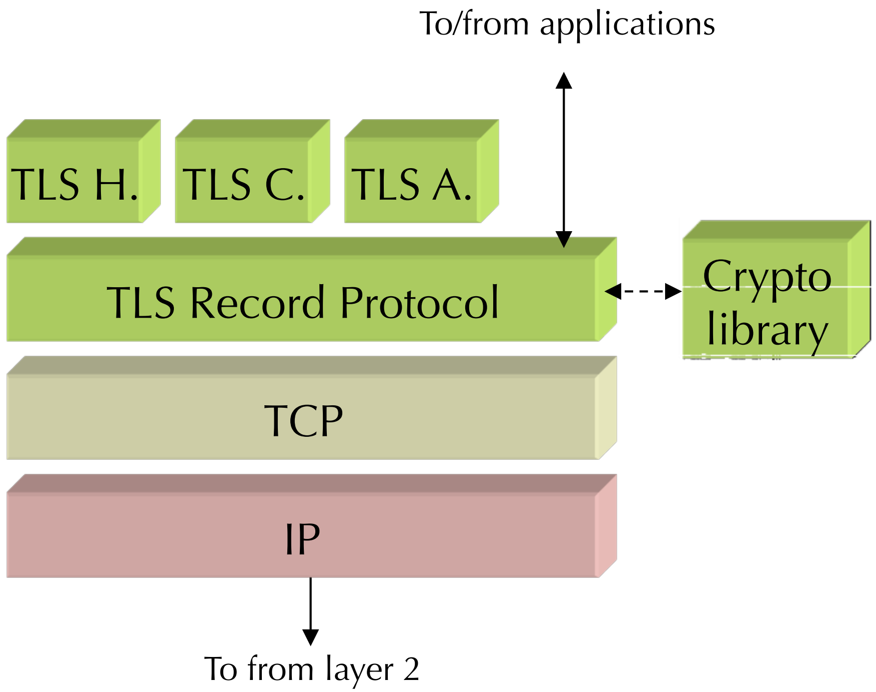
- *Transport Layer Security* is a protocol suite defined to offer security to applications: it works between TCP or UDP and applications
- Security features
  - Authentication
  - Integrity protection
  - Confidentiality
- TLS' goal is to provide a secure, authenticated channel based on ephemeral keys derived from longer cryptographic credentials of several kinds:
  - X.509 certificates
  - PSK
  - PSK/Kerberos
  - OpenPGP keys
  - Secure Remote Passwords (SRP)
- Two protocol variants
  - [RFC 4346] TLSv1.1, works above TCP
  - [RFC 4347] DTLSv1.0, works above UDP
- Here we will just see TLS (v1.0)

TLS: historical notes

- The story so far:
  - 1995: *Secure Socket Layer* (SSL) v2, Netscape Inc., protection of HTTP sessions
  - 1997: SSLv3, bug fixes, the most widely used variant to date
  - 1999: TLS (Transport Layer Security) v1.0, SSLv3 standardized by IETF, supports DSS along with RSA (works around the RSA patent), plus other minor differences
  - 2005/2006: TLS v1.1 e DTLS v1.0, minor differences from TLS v1.0, DTLS works on top of UDP [RFC 4347], support for PSKs [RFC 4279], support for OpenPGP keys, SRPs, etc.
- We will focus on SSLv3/TLSv1.0
  - TCP only (no UDP)
  - Long term credentials: DSS or RSA keypairs, i.e., X.509 certificates (no PSK)

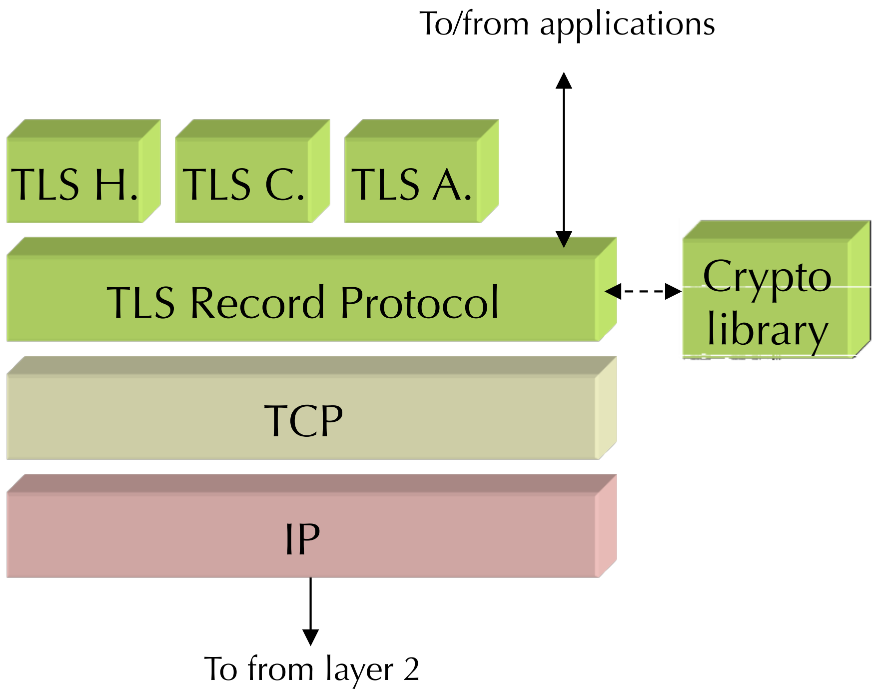
The TLS architecture
TLS Record Protocol

- Same services to above layers as TCP, with the added security
- Transports either user data, or other TLS protocol messages
- Once the TLS session is up and running (ephemeral keys have been installed), anything that the record protocol transports is protected (integrity, authentication, confidentiality)



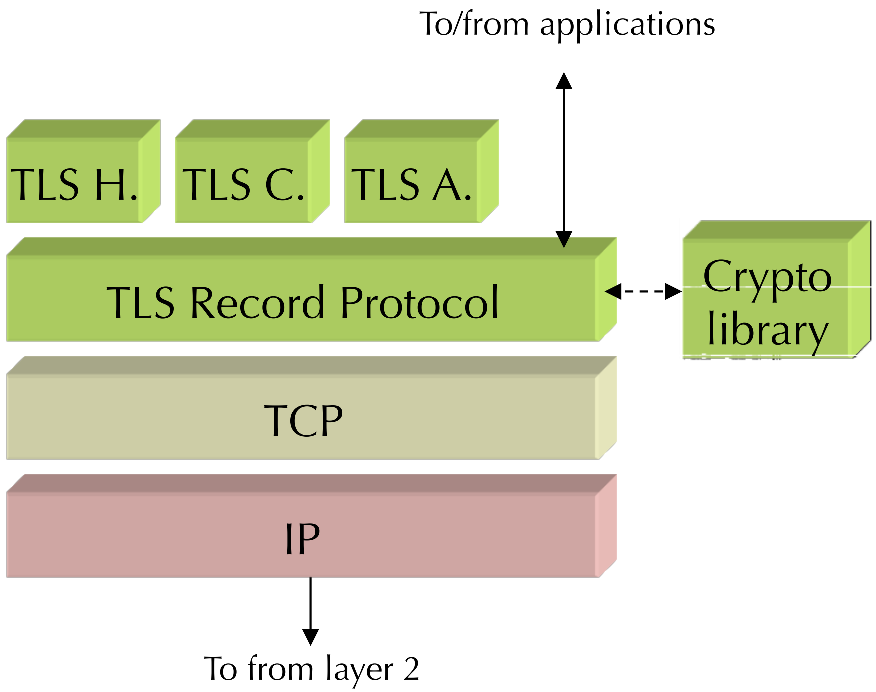
The TLS architecture
TLS Handshake Protocol

- Protocol used to manage security all parameters, such as cipher suite, ephemeral keys, etc., and handle authentication
- Deriving ephemeral keys is expensive, so TLS introduces the concept of ***session***, over which multiple ***connections*** can be setup



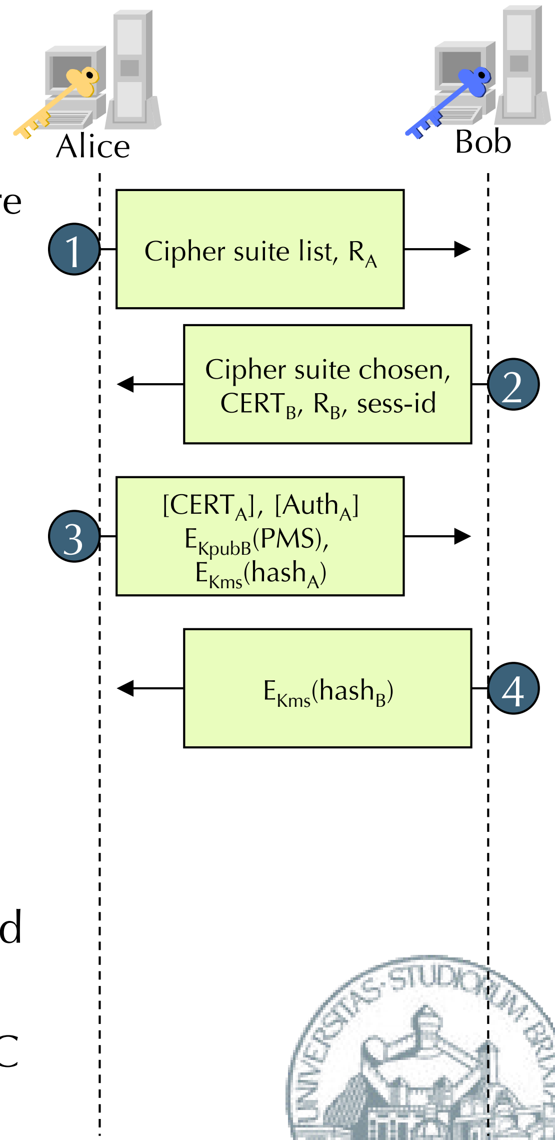
The TLS architecture
TLS Change CipherSpec and *Alert Protocol*

- Change CipherSpec Protocol
  - A really, really simple protocol: 1-bit message, signaling the end of the cleartext part of the handshake
- Alert Protocol
  - Used to signal error conditions, for example when there are problems with the underlying transport protocol



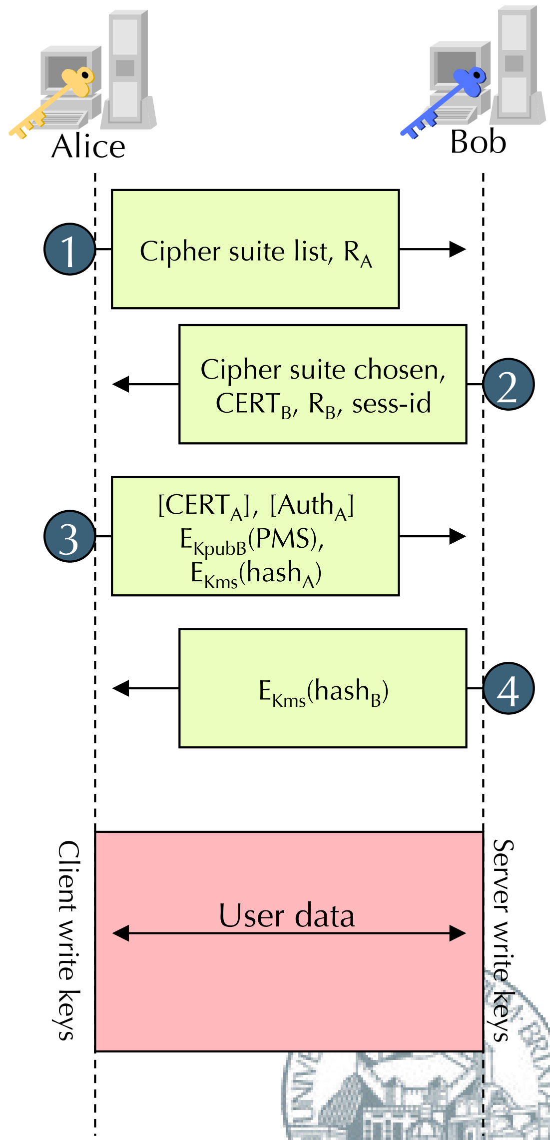
TLS Handshake protocol: high level overview

- Goals
  - Authentication of B to A and, optionally, of A to B
  - Setup of ephemeral ***session*** key K_{ms} (*Master Secret*), as a "seed" for one or more ***connection*** keys
- In TLS, A (*initiator*) is the ***client***, while B (*responder*) is the ***server***
- The Handshake protocol messages are transported by the TLS Record protocol
- Each record message can contain more than one handshake messages
  - For example, record message (2) usually contains messages "server_hello, certificate, [certificate_request], server_hello_done"
- $R_{A,}$ R_B: random numbers
- $Auth_A = SIG_{KprivA}(hash(previous\ messages))$
- PMS: pre-master secret, random number chosen by A
- $K_{ms} = f_{PMS}(R_A, R_B)$, where f is a MAC function derived from both SHA1 and MD5
- $hash_{A/B} = g_{Kms}(\{client/server\}, previous\ messages)$, where g is another MAC function derived from both SHA1 and MD5



TLS Handshake protocol

- At this point we have a ***TLS session***
  - Authentication [optionally mutual]
    - A authenticates B by the fact that B can prove they are able to decrypt PMS (i.e., calculate the correct K_{ms}) with the private key associated to $CERT_B$
    - Optionally, B authenticates A through $Auth_A$
  - TLS session, identified by (K_{ms}, sess-id). K_{ms} is a 384 bit (48 byte) string
- One or more ***TLS connections*** can now be instantiated, by deriving the necessary ephemeral keys from K_{ms}. User traffic will be protected by these TLS connections inside the TLS record protocol
- By key expansion, ***three key pairs*** are derived from K_{ms}
  - Client write MAC, client write, client IV (K_{cm}, K_c, IV_c)
  - Server write MAC, server write, server IV (K_{sm}, K_s, IV_s)
  - Key expansion is similar to the one used to derive K_{ms} from PMS, i.e., it is based on a MAC function that works on K_{ms}, R_A, R_B
  - Note that there are three keys ***for each of the two directions***

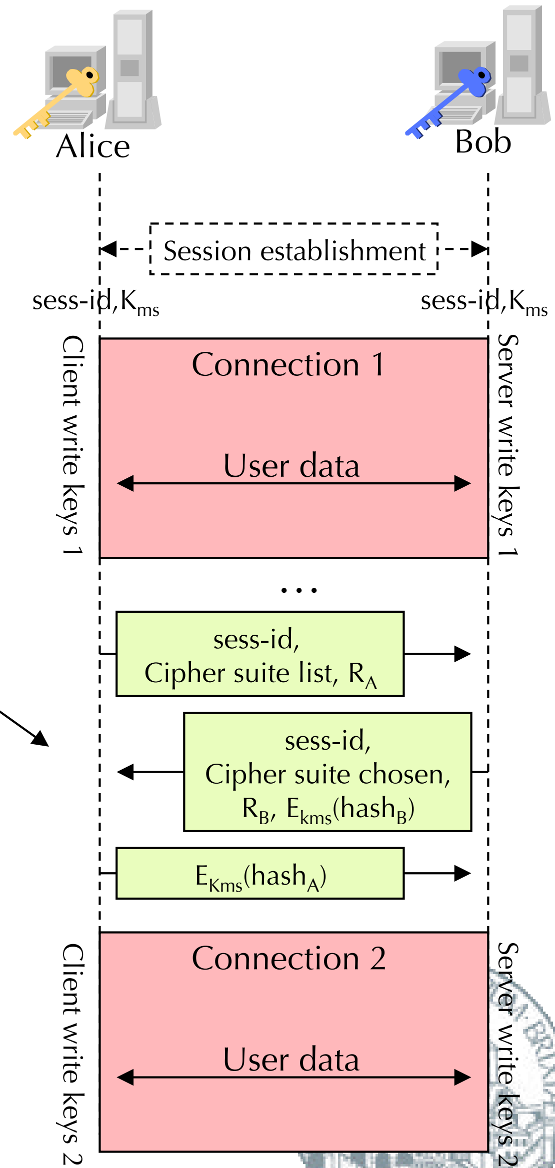


TLS Handshake protocol: notes

- PFS is optional: K_{ms} can optionally be derived from a PMS=DH key
- CA hierarchies
  - When A or B send a certificate, they can also include ***chains of (CA) certificates***
  - When B request that A authenticates with their certificate, usually B includes in the request a list of ***known and admissible CAs***

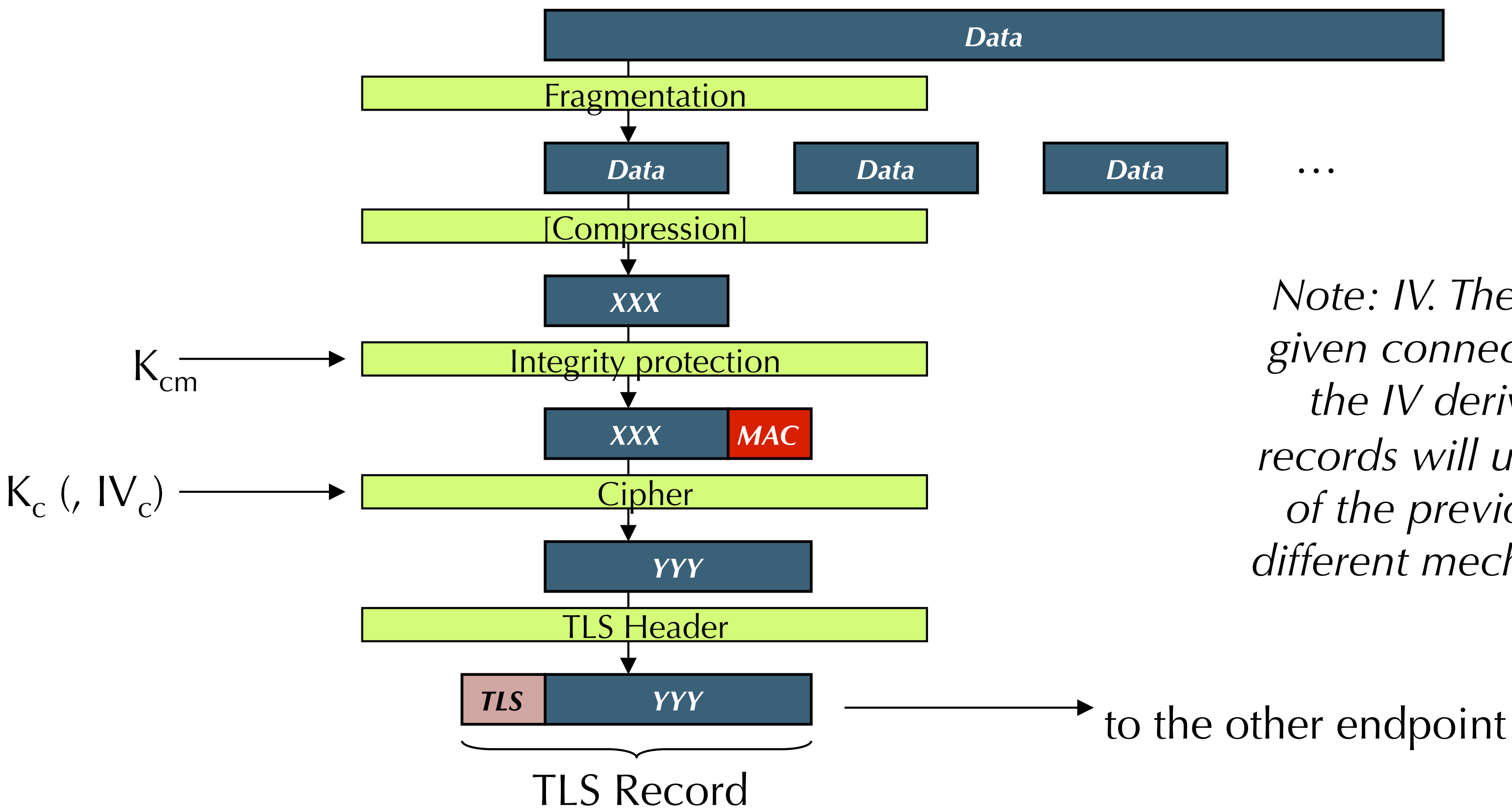
Session/connection: *session resumption*

- Once a TLS session is in place, A and B can derive multiple sessions without going through the initial exchange, i.e., without having to derive a new K_{ms}
- This variant is used to continue a session (actually, open another connection) for which we still have a valid pair (sess-id, K_{ms})
- Cipher suite can be re-negotiated
- R_A and R_B are re-negotiated



Encrypting data

- TLS Record protocol, once a TLS connection is active, cryptographically protects all user data and all subsequent TLS messages (e.g., TLS Alert protocol)
- Compression is optional
- MAC = HMAC in TLS, HMAC-like in SSLv3



Note: IV. The first TLS Record in a given connection will need to use the IV derived from K_{ms}. Next records will use as IV the last bytes of the previous record (v1.0). A different mechanism is used in v1.1.

TLS cipher suites

- Contrary to IKE, SSLv3 and TLS define a finite and pre-set number of cipher suites
- As usual, each cipher suite defines algorithms, modes of operation, parameters, etc.
  - For example: 3DES-CBC + HMAC-MD5 + SHA1, etc.
- Client tells Server what cipher suites it supports, in descending order of preference
- The Server selects one, usually the one that it supports that is highest on the list
- Warning: never use “EXPORTABLE” cipher suites (40 bit ephemeral keys!)

TLS Structure

- Handshake protocol
  - Establish shared keys (typically using public key cryptography)
  - Negotiate algorithms, modes, parameters
  - Authenticate one or both sides
- Record protocol
  - Carry individual messages
  - Protected under symmetric keys
- This is a common design (SSH, IPsec, etc.)

TLS 1.2: RSA Handshake Skeleton

Client $\qquad\qquad$ Server

ClientHello [Random] $\longrightarrow$

ServerHello [Random], Certificate $\longleftarrow$

E(K_s, Master Secret), *Finished=MAC(MS, Handshake)* $\longrightarrow$

Finished=MAC(MS, Handshake) $\longleftarrow$

Application data $\longleftrightarrow$

TLS 1.2: (EC)DHE Skeleton

Client   Server

ClientHello [Random] $\longrightarrow$

$\longleftarrow$ ServerHello [Random], Certificate, Sign(K_s, g^s, ...)

g^c, *Finished* $\longrightarrow$

$\longleftarrow$ *Finished*

$\longleftrightarrow$ *Application data*

TLS 1.2: (EC)DHE + Client Authentication

Client		Server
	ClientHello [Random] $\longrightarrow$	
	$\longleftarrow$ ServerHello [Random], Certificate, CertificateRequest, Sign(K_s, g^s, ...)	
	g^c, Certificate, Sign(K_c, ...), *Finished* $\longrightarrow$	
	$\longleftarrow$ *Finished*	
	$\longleftrightarrow$ *Application data*	

More on Negotiation

- ClientHello contains more than just random values

```
struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites<2..2^16-2>;
    CompressionMethod compression_methods<1..2^8-1>;
    select (extensions_present) {
        case false:
            struct {};
        case true:
            Extension extensions<0..2^16-1>;
    };
} ClientHello;
```

Client Offers, Server Chooses

```
struct {
    ProtocolVersion server_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suite;
    CompressionMethod compression_method;
    select (extensions_present) {
        case false:
            struct {};
        case true:
            Extension extensions<0..2^16-1>;
    };
} ServerHello;
```

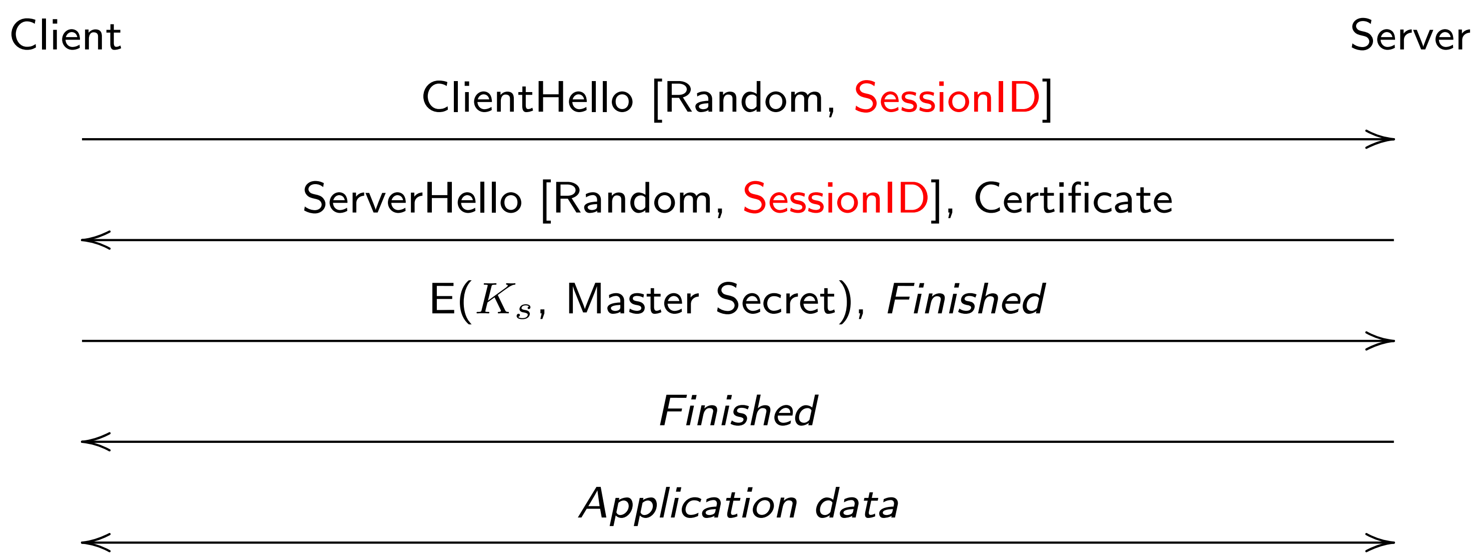
What's in a Cipher Suite?

- Key Exchange (RSA, DHE, ECDHE, PSK, ...)
- Authentication (RSA, DSS, ECDSA, ...)
- Encryption (AES, Camellia, ...)
- MAC (MD5, SHA1, SHA256, ...)

Session Resumption

- "Public key" operations are comparatively expensive
  - They used to be *really* expensive
- Solution: amortize this operation across multiple connections

Session Establishment



Session Resumption

Client Server

ClientHello [Random, SessionID] $\longrightarrow$

$\longleftarrow$ ServerHello [Random, SessionID], *Finished*

Finished $\longrightarrow$

$\longleftarrow$ *Application data* $\longrightarrow$

- No new public key operations
- Reuse MS from last handshake

TLS 1.3 Objectives

- *Clean up:* Remove unused or unsafe features
- *Security:* Improve security by using modern security analysis techniques
- *Privacy:* Encrypt more of the protocol
- *Performance:* Our target is a 1-RTT handshake for naive clients; 0-RTT handshake for repeat connections
- *Continuity:* Maintain existing important use cases

Removed Features

- Static RSA
- Custom (EC)DHE groups
- Compression
- Renegotiation*
- Non-AEAD ciphers
- Simplified resumption

*Special accommodation for inline client authentication

Removed Feature: Static RSA Key Exchange

- Most SSL servers prefer non-PFS cipher suites [SSL14] (specifically static RSA)
- Obviously suboptimal performance characteristics
- No PFS
- Gone in TLS 1.3
- Important: you can still use RSA certificates
  - But with ECDHE or DHE
  - Using ECDHE minimizes performance hit

Removed Feature: Compression

- Recently published vulnerabilities [DR12]
- Nobody really knows how to use compression safely and generically
  - Sidenote: HTTP2 uses very limited context-specific compression [PR14]
- TLS 1.3 bans compression entirely
  - TLS 1.3 clients MUST NOT offer any compression
  - TLS 1.3 servers MUST fail if compression is offered

Removed Feature: Non-AEAD Ciphers

- Symmetric ciphers have been under a lot of stress (thanks, Kenny and friends)
  - RC4 [ABP+13]
  - AES-CBC [AP13] in MAC-then-Encrypt mode
- TLS 1.3 bans all non-AEAD ciphers
  - Current AEAD ciphers for TLS: AES-GCM, AES-CCM, ARIA-GCM, Camellia-GCM, ChaCha/Poly (coming soon)

Removed Feature: Custom (EC)DHE groups

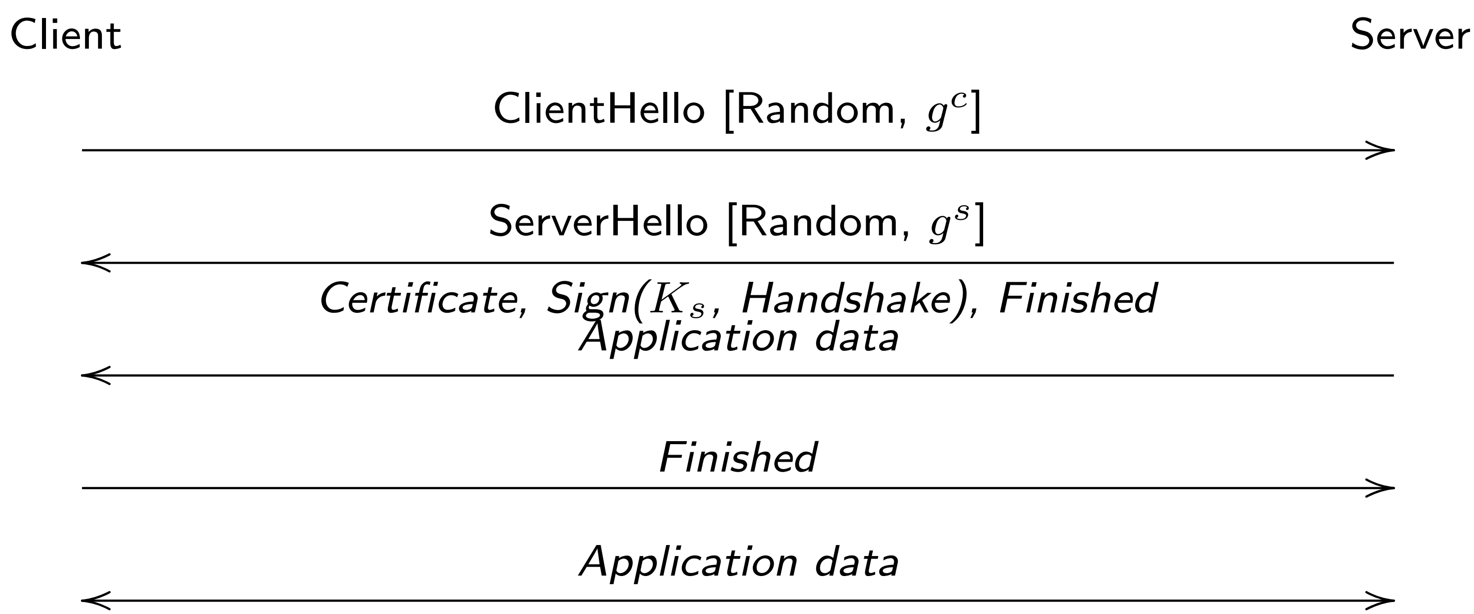
- Previous versions of TLS allowed the server to specify their own DHE group
  - The only way things worked for finite field DHE
  - (Almost unused) option for ECDHE
- This isn't optimal
  - Servers didn't know what size FF group client would accept
  - Hard for client to validate group [BLF+14]
- TLS 1.3 only uses predefined groups
  - Existing RFC 4492 [BWBG+06] EC groups (+ whatever CFRG comes up with)*
  - New FF groups defined in [Gil14]

*Bonus: removed point format negotiation too

Optimizing Through Optimism

- TLS 1.2 assumed that the client knew nothing
  - First round trip mostly consumed by learning server capabilities
- TLS 1.3 narrows the range of options
  - Only (EC)DHE
  - Limited number of groups
- Client can make a good guess at server's capabilities
  - Pick its favorite groups and send a DH share

TLS 1.3 1-RTT Handshake Skeleton



- Server can write on its first flight
- Client can write on second flight
- Keys derived from handshake transcript through server MAC
- Server certificate is encrypted
  - Only secure against passive attackers

TLS 1.3 1-RTT Handshake w/ Client Authentication Skeleton

Client → Server

ClientHello [Random, g^c] $\longrightarrow$

$\longleftarrow$ ServerHello [Random, g^s]

$\longleftarrow$ *CertificateRequest, Certificate, Sign(K_s, Handshake), Finished*
Application data

Certificate, Sign(K_c, Handshake), Finished $\longrightarrow$

$\longleftrightarrow$ *Application data*

- Client certificate is encrypted
- Secure against an active attacker
- Effectively SIGMA [Kra03]

What happens if the client is wrong?

- Client sends some set of groups (P-256)
- Server wants another group (P-384)

Client → Server: ClientHello [Random, DH(P256)]

Server → Client: HelloRetryRequest [P384]

Client → Server: ClientHello [Random, DH(P256), DH(P384)]

...

- This shouldn't happen often because there are a small number of groups
  - Client should memorize server's preferences

0-RTT Handshake

- Basic observation: client can cache server's parameters [Lan10]
  - Then send *application data* on its first flight
- Server has to *prime* the client with its configuration in a previous handshake

TLS 1.3 0-RTT Handshake Skeleton

Client Server

ClientHello [Random, g^c, server_configuration=XXX]

$\longrightarrow$

Application data

ServerHello [Random, g^s]

$\longleftarrow$

Certificate, Sign(K_s, Handshake), Finished

Application data

$\longleftarrow$

Finished

$\longrightarrow$

Application data

$\longleftrightarrow$

Anti-Replay

- TLS anti-replay is based on each side providing random value
  - Mixed into the keying material
- Not compatible with 0-RTT
  - Client has anti-replay (since they speak first)
  - Server's random isn't incorporated into client's first flight

Anti-Replay (borrowed from Snap Start)

- Server needs to keep a list of client nonces
- Indexed by a server-provided context token
- Client provides a timestamp so server can maintain an anti-replay window

Traffic Analysis Defenses

- TLS 1.2 is very susceptible to traffic analysis
  - Content "type" in the clear
  - Packet length has minimal padding
    - 0-255 bytes in block cipher modes
    - No padding in stream and AEAD modes
- TLS 1.3 changes
  - Content type is encrypted
  - Arbitrary amounts of padding allowed
  - ... but it's the application's job to set padding policy

Packet Format

TLS 1.2 Packet Layout

TLS 1.3 Packet Layout

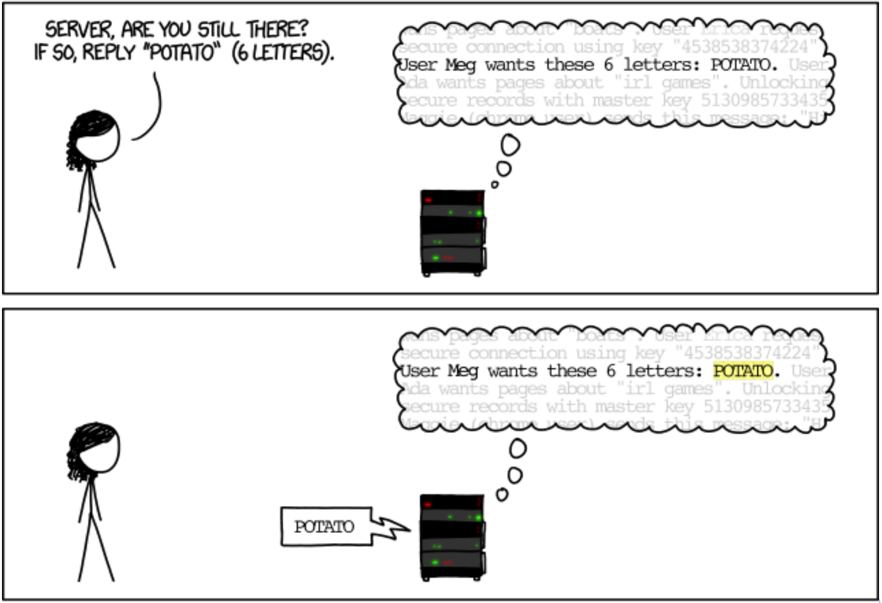
The Heartbeat extension to TLS

- designed to enable a low-cost, keep-alive mechanism
- so that client and server know that they're still connected and all is well
- described in RFC 6520 for TLS and DTLS
- supported in OpenSSL v1.0.1, enabled by default

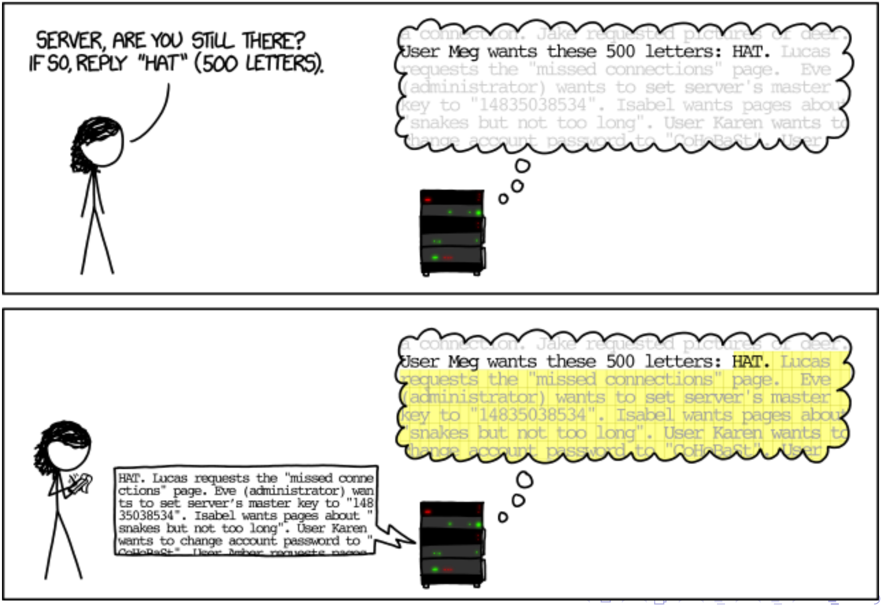
Protocol

1. client sends a a packet of type heartbeat_ request, along with an arbitrary payload and a field that defines the payload length.
2. server answers with a heartbeat_response that contains an exact copy of the payload.

Heartbeat correct execution



Heartbleed attack



Lesson learnt

The Heartbleed vulnerability looks really trivial...

- it's essentially a buffer overread vulnerability, where inadequate bounds-checking is carried out at runtime
- a client should not trust the payload length presented in the heartbeat_request packet
- more generally, placing trust in user-supplied input is often a bad idea
- also...a lot can go wrong between design and implementation