# Secure Channel

- TCP / IP by itself doesn't provide authenticity / confidentiality
- Need cryptography to layer a secure channel on top of it
  - TLS / SSL

## Public Key Operations

- `KeyGen()`: generates public key `PK` and secret key `SK`
- `Encrypt(PK, msg m)`: generates ciphertext `c`
- `Decrypt(SK, c)`: gives back `m`
- `Sign(SK, msg m)`: generates signature `sig`
- `Verify(PK, m, sig)`: is signature valid?

## Private / Symmetric Key Operations

- `KeyGen()`: generates private / symmetric key `K`
- `Encrypt(K, m)` generates `c`
- `Decrypt(K, c)` generates `m`
- `MAC(K, m)` generates a tag `t`
  - Should use diff key from encryption
- Very fast, but key has to be a shared secret

## DH Key Exchange

- If you have `PK_A, SK_A, PK_B, SK_B` there is a way to combine `PK` and `SK` such that:
  - `shared = DH(PK_A, SK_B) = DH(PK_B, SK_A)`
- Useful for computing shared secrets without having to send them over the wire!

## Strawman 1:

```
C --> S: connection request
C <-- S: client receives server's public key PK_S
C --> S: client sends a new key to server to use in their communication
C <-> S:
```

Problem: authenticating the server

- What if an adversary intercepts messages to the server?
- The client then has a shared key with the adversary
- The adversary can connect to the real server and relay messages
- "Man in the middle" attack
- Idea: need to bind PK_S to server's identity

## Certificates

Solution: trusted authority server

- Maintains table of (principal name, public key) pairs
  - Principal name is typically the server's DNS name
  - Choice of name must capture the client's intended communication target
  - Crucial that it is correct
- Everyone needs to know the trusted authority server's public key

- Each person could query the server, but that's bad for availability / performance
  - Authority server can instead sign a message: `Sign(SK_CA, {server, PK_server})`
  - This is the "certificate"
  - Clients can verify it is correctly signed by the certificate authority because they all have the CA's public key

## Revised Protocol

```
C --> S: connection request
C <-- S: PK_S and server's certificate (Sign(SK_CA, { server, PK_server})
C --> S: encrypted fresh key
C <-> S: communication with fresh key
```

Alternative:

- Trust-on-first use:
  - Assume first connection is OK
  - Remember binding between PK_S and server's name for future use

Problem: authenticating the messages

- When a client sends a message, an adversary can modify it
- Decrypt always succeeds, so adversary can tamper with ciphertext
- This can sometimes have predictable effects (i.e. changing a bit flips a bit in message)
- Solution: use MAC to compute an authentication tag
  - Use two keys: one for encrypting and one for MAC
  - When you send ciphertext, also send the `t = MAC(MAC_KEY, c)`
  - If the tag is incorrect, something was modified

---

- New problem: replay attacks
  - Two common solutions:
  - Sequence number in message
    - Receiver tracks last-seen SN and discards messages with older ones
    - Good for stream oriented protocols
  - Expiration / session ID in message
    - Receiver tracks all past messages sent within expiration time / session
    - Checks to see if any have been previously received
    - Good for message-oriented protocols
- Another problem: server breakins
  - What if a bad guy later compromises the server?
  - Can use SK_S to decrypt old network traffic
    - Even if you use a fresh key for every communication, you sent that across the stream at some point

## Forward Secrecy

- Solution: don't use long-term keys for encryption
  - Subsceptible to decryption by a future adversary
- Add a long-term signing key
- Use short-lived keys for encryption and long-term keys for signing
  - Future adversary can forge signatures in the future, but it's too late at that point
- New protocol:
  - Both C and S generate connection key pairs taht will be combined with DH

```
C --> S: connection request with PK_conn_C
C <-- S: PK_conn_S, Sign(SK_CA, {server: PK_S}), Sign(SK_S, PK_conn_S)

Both can then compute PK_conn
```

```
C --> S: Encrypt(PK_conn, freshKey)
C <-> S: Encrypt(freshKey, m)
```

- The SK_conn and PK_conn are only used once to get the fresh key and then are thrown away

# SSL / TLS

- Secure channel protocol for the web
- TLS is the international standard version of SSL
- Main parts:
  - Record protocol
  - Handshake protocol
- Threat model:
  - Active attackers that tamper and forge messages
  - Passive attackers that eavesdrop
- SSL 2.0, SSL 3.0 (1996), TLS 1.0 (1999), TLS 1.1 (2006), TLS 1.2 (2008), TLS 1.3 (2018)

---

- Issue that complicates TLS: figuring out what encryption scheme to use
  - Cipher negotiation (how the shared secret will be used)
  - Key exchange (DH, figuring out how to combine the keys)
  - Signing (how to compute these MAC things)
- Big problem with TLS: reducing round trips

Naive version (+2 RTT (round trip time), TLS 1.2):

```
C --> S: hello with all supported encryption schemes
S --> C: tells the client what to use

Then we spend a round trip doing above to actually get the shared secret
```

TLS1.3: client guesses the key exchange algorithm when sending the first message to include PK_conn_C

- If correct, server can immediately send back PK_conn_S
- Web browsers can remember which servers use which schemes
- Almost everyone just uses X25519 as the default and it just works

0 RTT: just making the guess correct is not enough to get 0 RTT because you still need the server to send back its public key

- CANNOT achieve forward secrecy
- But you can turn on this mode if you have already talked to the server before and remember which K to use
- Ways of implementing this:
  - Server stores sessions and keys used for those sessions
    - Client just has to send the session ID and the server remembers
    - Server has to store state
  - Stateless version: server can store a K_ticket globally and whenever the server and client have a communication, it encrypts the key with K_ticket and gives it back to the client
    - The client just has to present this ticket again and the server can remember what key to use
- Subsceptible to replay attacks because we are reusing these keys and not generating fresh ones

## Downgrading Attack

- Adversaries can modify the hello messages (since they aren't signed) and force the parties to use a weaker cipher, which might have been discovered to have vulnerabilities
- TLS 1.3 solves this by when the signature is sent back, we sign the entire transcript of the conversation we have been having

# E2E

- Compromised server can still access data if the server has decryption
  - Alternative: do not give server the decryption key
- Data encrypted with key of intended recipient
  - Not necessarily server that transmits / stores message
  - Server stores / forwards ciphertext
- How to deal with shared encrypted data (i.e. multiple users have access to shared doc)
  - Naive plan: encrypt same document with each user's key
  - Downside: multiple copies that are potentially divergent
  - Better plan: chain keys
    - Generate a new key just for the document
    - Encrypt document with the document's public key
    - Encrypt document's secret key with each user's public key