6.033: Security - Secure Channels
Lecture 22
Katrina LaCurts, lacurts@mit.edu

```
**************************************************************************
* Disclaimer: This is part of the security section in 6.033. Only        *
* use the information you learn in this portion of the class to          *
* secure your own systems, not to attack others.                         *
**************************************************************************
```

0. Today's threat model
   - Last time: adversary with access to server
   - Today: adversary in the network
   - What can adversary in the network do?
     - Observe packets
     - Corrupt packets
     - Inject packets
     - Drop packets
   - Some can be combated with techniques you already know
     - TCP senders retransmit dropped packets
     - Corrupt packets get dropped (at a router, usually), and thus also retransmitted
   - Need a plan, though, for carefully corrupted, injected, or sniffed packets
   - This lecture: focus on preventing an adversary in the network from observing/tampering with contents of packets
   - Goals (policy)
     1. Confidentiality: adversary cannot learn message contents
     2. Integrity: adversary cannot tamper with message contents
     - More accurately, if the adversary tampers with the message contents, the sender and/or receiver will detect it.
   - Result is known as a "secure channel"

1. Secure Channel Primitives
   - Ensure confidentiality by encryption
     - Encrypt(k, m) -> c ; Decrypt(k, c) -> m
       - k = key (secret; unknown to adversary, never transmitted)
       - m = message
       - c = ciphertext
       - Property: given c, it is (virtually) impossible to obtain m without knowing k
     - Encryption alone does not provide integrity
       - Adversary could change some bits in ciphertext
       - Other mathematical reasons
         - See http://security.stackexchange.com/questions/33569/why-do-you-need-message-authentication-in-addition-to-encryption
         - Section 11.4.4 of the course textbook.
   - Ensure integrity via message authentication codes (MAC)
     - MAC(k, m) -> t

    - k = key
    - m = message
    - t = output
  - Similar to hash functions. Difference: uses a key
    - Alternate name: "keyed hash function"
    - Adversary can't compute the MAC of a message; needs key. (This is not true for regular hash functions)
    - There are other subtle differences we won't get into. One example: MACs are not always subject to the same mathematical requirements as cryptographic hash functions.

2. Secure Channel Abstraction
   - (See slide for diagram)
   - If adversary intercepts [c|h] and tampers with it, receiver will know; MAC won't check out.
   - Problem: adversary can intercept, and then retransmit message ("replay" message)
   - Solution: Include a sequence number in every message, and choose a new random sequence number for every connection
   - If adversary intercepts message, can't replay in the same way because sender won't reuse sequence number
     - Assume sequence numbers don't wrap around
     - Aside: In reality, if there is a conversation that is long enough that the sequence number space is exhausted, a session is "renegotiated" between the sender and the receiver. (You could, for instance, imagine that whenever a session is renegotiated, the sender and receiver both change their keys. In reality, they change a particular random value known as the session ID.
   - But if receiver is also sending to the sender (i.e., if they're both sending), the receiver might use that sequence number. So adversary could replay in the other direction (a "reflection" attack)
   - Solution: Use different keys in each direction

3. Key Exchange
   - How do sender/receiver get keys in the first place? Can't just send them in the clear in the beginning
   - Diffie-Hellman key exchange
     - Two parties: Alice and Bob ("sender" and "receiver" before)
     - Alice and Bob pick:
       - a prime number p
       - a "generator" g
         - Aside: For g to be a generator, it has to be a "primitive root modulo p". If you want to know more about primitive roots, take a cryptography, number theory, or abstract algebra class. They're all good!
       - p and g don't need to be secret; assume adversary knows them
     - Alice picks random number a (secret)
     - Bob picks random number b (secret)

    - Alice sends g^a mod p to Bob
    - Bob sends g^b mod p to Alice
    - Alice computes (g^b mod p) ^ a mod p = g^ab mod p
    - Bob computes (g^a mod p) ^ b mod p = g^ab mod p
    - Secret key = g^ab mod p
    - Adversary can learn p, g, g^a mod p, g^b mod p. From this, one cannot calculate g^ab mod p; you need to know either a or b to do that.
      - Trust me on that; won't prove it in 6.033
  - Problem: on-path attacker
    - Adversary in middle of network intercepts (and responds to) messages in both directions; Alice thinks she has established a connection with Bob, and vice versa; in reality, they've both established a connection with the adversary.

4. Cryptographic Signatures for Message Authentication
  - Problem with the above is that messages aren't authenticated; Alice doesn't know if she's really talking to Bob, and vice versa
  - Before: shared key between the two parties. Known as symmetric key cryptography.
  - For signatures: public-key cryptography
    - Each user generates a key pair: (PK, SK)
    - PK is public: known to everyone, adversaries included
    - SK is secret: known only to user
    - PK and SK are related mathematically; we will not get into that here
      - RSA is a scheme that generates a key-pair for you.
    - SK let's you sign messages; PK let's you verify signatures (but NOT perform the signing)
  - Primitives
    - Sign(SK, m) -> sig.
      - SK = secret key
      - m = message
      - sig = signature
    - Verify(PK, m, sig) -> yes/no.
      - PK = public key
      - m = message
      - sig = signature
      - "yes/no" -> yes if signature is verified, no otherwise
  - This is all similar to MACs. Signatures don't require parties to share a key

5. Key Distribution
  - How do we distribute public keys? Lots of ideas
  1. Alice remembers the key she used to communicate with Bob the last time.
     - Easy to implement, effective against subsequent man-in-the-middle attacks
     - Doesn't protect against MITM attacks the first time around, doesn't allow parties to change keys

2. Consult some authority that knows everyone's public key
   - Doesn't scale (client asks authority for a PK for every new name)
   - Alice needs server's public key beforehand
3. Authority, but pre-compute responses. Authority creates signed messages: {Bob, PK_bob}_{SK_as}. Anyone can verify that the authority signed this message, given PK_as. When Alice wants to talk to Bob, she needs a signed message from the authority, but it doesn't matter where this message comes from as long as the signature checks out (i.e., Alice could retrieve the message from a different server).
   - This signed message is a certificate
   - More scalable

- Certificate authorities bring up questions:
  - Who should run the certificate authority?
  - How does the browser get this list of CAs?
    - Generally they come with the browser.
  - How does the CA build its table of names <-> public keys?
    - Have to agree on how to name principals, and need a mechanism to check that a key corresponds to a name
  - What if a CA makes a mistake?
    - Need a way to revoke certificates
    - Expiration date? Useful in long term, not for immediate problems
    - Publish certificate revocation list? Works in theory, not as well in practice (CRLs sometimes incorrect, not always updated immediately)
      - Aside: http://ssl-research.ccs.neu.edu/papers/Heartbleed-IMC.pdf
    - Query online server to check certificate freshness? Not a bad idea
- Alternative: avoid CAs by using public keys as names (protocols: SPKI/SDSI). Works well for names that users don't have to remember/enter

6. TLS: A protocol that does all of this
   - Lots of parts to this protocol; its complexity can cause problems (frequently implemented incorrectly)
   - Notice that client/server use public-key crypto to exchange a secret, which they use to generate keys for symmetric crypto
     - Symmetric crypto is much faster than public-key crypto

7. Discussion
   - Why isn't traffic encrypted by default?
     - Can be computationally expensive
     - Complex to implement
     - Wasn't a well-known thing for most users until relatively recently
       - Historically just applied to transactions that obviously need to be secured. E.g., banking

  - Maybe we're at a point now where these arguments no longer apply?
- Open vs. Closed Design
  - Should system designers keep details of encrypt/decrypt/MAC/etc. a secret?
  - No: make the weakest practical assumptions about the adversary. Assume they known the algorithms, but not the secret keys.
    - Also lets us reuse well-tested, prove algorithms
    - Plus, if key is compromised, we can change it (unlike the algorithms)