Benjamin Fenster
CS-266 (Ling)
19 December 2005

Notes – Authentication

I. Humans vs. Computers
   a. Humans can remember only short keys
   b. Computers can authenticate to each other using longer (more secure) keys that are totally random
   c. Passwords as Keys
      i. Could use for things like DES: Use a password directly as the key
      ii. For RSA, cannot remember the 512 bit key. So encrypt the private key with a password and store it in a file somewhere
      iii. Would be nice to use the password directly
      iv. Could use the password as a seed for the random number generator – you'd get the same primes every time since they're just a sequence
      v. The problem is that it's hard to do all the exponentiating for finding primes
      vi. Plus, would need to agree on which random number generator to use (there are many)
      vii. Could remember which 'indices' in the sequence (197 tries, …) but people won't want to bother remembering
   d. Eavesdropping
      i. Could use challenge-response
         1. Server sends random R
         2. User signs with a private key
         3. This is secure if the server is compromised: the server stores no secret information
         4. It's secure against eavesdropping, since the challenge will be different every time
      ii. With a secret key crypto system
         1. User encrypts $E_k(R)$ using the key shared between the user and server
         2. If the server were compromised, Trudy would get access to the key!
         3. It's still safe from eavesdropping, since Trudy sees only a plaintext / ciphertext pair.
      iii. Using a hash
         1. Server stores only a hash of the password
         2. If the password is sent in the clear, eavesdropping would give it away

II. Trusted Intermediaries
   a. Hard to remember secret keys for *everybody*(there are too many keys)
   b. Key Distribution Center authenticates both parties
   c. Generates a per-session key on the fly
   d. Then everybody needs to remember only one key for the KDC
   e. Problems
      i. This becomes a single point of failure. If the KDC is compromised then there's no security at all
      ii. Could also be a performance bottleneck
   f. Multiple KDCs
      i. Since there's more than one KDC, they need to communicate with each other
      ii. I want to talk to someone listed in the KGB's KDC.
      iii. Send a message to the local CIA KDC, and it creates a key to communicate with the KGB's KDC.
         1. The CIA sends a key back to you and forwards it to the KGB.
         2. Requires having a key between the CIA and KGB
      iv. Then use that key to make a request of the KGB's KDC
      v. Must store pairwise keys among KDCs.
   g. Certificate Authority
      i. Ensures that you're getting the person's real public key

    ii. The CA signs public keys, so as long as you know the CA's public key you can verify the signature and thus verify anybody's public key.
    iii. Compromised CA could hurt future security but old public keys are still good
    iv. There's no need to contact the CA to verify that a key is signed correctly though, so there's no bottleneck and there's no risk of eavesdropping from a compromised certificate authority
    v. Certificates are valid forever unless there's a mechanism to invalidate them after a time. A revocation list would accomplish this (a blacklist of invalid certificates)
    vi. Having multiple certificate authorities just requires signing one another's public keys

III. Passwords
  a. Password-based authentication
  b. Computer knows you are who you claim because you know a secret password
  c. Need to consider potential for eavesdropping (problem with dumb terminals and cell phone cloning)
  d. How to get the password initially?
    i. Hand it out? Probably hard to remember
    ii. Type it at a root terminal? Potentially gives away a few seconds of root access
    iii. Could set password equal to username initially, but that's insecure
  e. Password Guessing
    i. Offline
      1. Take the hashed version, try hashing dictionary words and comparing
      2. Prevented by choosing good passwords
    ii. Online
      1. Trying different passwords at login.
      2. Want to make this hard
      3. Have an increasingly long delay after an incorrect password
  f. Good Passwords
    i. 20 random digits
    ii. 11 characters of letters, digits, and punctuation
    iii. 16 characters of random, pronounceable "words"
    iv. A user=selected password has only about two useful bits per character, so need 32 characters to make it secure.
  g. Storing Passwords
    i. Store per node. Different passwords on different machines (not used anymore)
    ii. Authentication Storage Server: Send a request to the server, which distributes passwords as needed.
    iii. Facilitator: Server just says "yes" or "no"
    iv. Need to be sure the node requesting passwords is authentic
  h. Trojan Horses: Pretend to be the regular login screen and intercept passwords

IV. Address-Based Authentication
  a. .rhosts (rsh: remote shell)
  b. /etc/hosts.equiv – trused hosts system wide
  c. Threats: If you can break into one, you can break into all of them
  d. Address spoofing: Getting harder to do

V. Authentication Token
  a. Physical keys, magnetic cards
  b. Smart cards
    i. Challenge / response
    ii. The card has some computational ability; can respond to challenges
  c. Cryptographic Calculator: Displays the time encrypted; type that into a reader
  d. All require a PIN to unlock it to prevent people from just copying the device

VI. Biometrics
  a. Terminology
    i. FAR: False Acceptance Rate. Worst type of error

        ii. FRR: False Rejection. Inconvenient, but not security critical (can be reduced by allowing more tries to get it right)
   b. Want to eliminate “Unstable” population (always rejected)
   c. Fingerprint
        i. 1 to 5% FRR, 0.01 to 0.0001% FAR
        ii. 2 seconds, 800 – 1203 bytes
   d. Hand Geometry
        i. 0.2 FRR, FAR
        ii. Under three seconds, 9 bytes
   e. Retinal Scans
   f. Voice Recognition: Vulnerable to background noise and changes in voice due to things like illness
   g. Signatures: Not very reliable

VII. Lamport’s Hash
   a. Want to be safe at login from both eavesdropping and from a compromised database
   b. One idea
        i. Server stores only the hash of the password
        ii. Safe from reading from the database
        iii. For this to work, user must send password and let the server hash it
        iv. The bad guy can “hear” the password as it’s sent so he can pretend to be that user.
        v. Even if you send the hash over the network instead, now it’s just a plaintext password since that’s what’s stored in the server anyway
   c. Lamport’s Hash
        i. Safe from both eavesdropping and a compromised database
        ii. $hash^n$(pwd) means to hash the password repeatedly (n times)
        iii. For every user, store username, n, $hash^n$(pwd)
        iv. The workstation computes x = $hash^{n-1}$pwd) given the value of n provided by the server
        v. The server computes hash(x) and compares that to the database
        vi. Then replace the password in the database with x and decrement n
        vii. Observing x gains nothing for Trudy, since the next login will require one *less* hash and the whole point is nobody can realistically calculate the “inverse” hash
   d. A Problem
        i. Trudy pretends to be the server, sends n = some low number
        ii. The user will send $hash^{24}$(pwd), perhaps, which Trudy can then use to compute all hashes for $n \geq 25$.
        iii. IF the server says n = 125, just hash the thing 100 more times.
   e. Include a Salt
        i. Pick a random r = a salt value
        ii. Compute $hash^n$(p | r)
        iii. Helps prevent dictionary attacks
   f. When n gets small, just reset it and change the password
   g. Pencil and Paper
        i. Print out $hash^1$(pwd), $hash^2$(pwd), . on paper.
        ii. Each time you login, type the password next on the list and cross it off
        iii. This works with computers that can’t locally compute a hash (dumb terminals)
        iv. It’s obviously vulnerable to somebody stealing the whole list

VIII. Mediated Authentication
   a. Needham Schroeder
        i. Alice sends N_1, “I want Bob” to KDC
        ii. KDC replies K_A {N_1, “Bob”, K_{AB}, ticket}
            1. N confirms that it’s really the KDC
            2. “Bob” confirms who’s being sought
            3. The fact that K_{AD} is used confirms that Alice is the other party

        4. Ticket = $K_B\{K_{AB}$, "Alice"$\}$
    iii. A forwards the ticket to Bob (which only he can decode) and sends the challenge $K_{AB}\{N2\}$
    iv. Bob replies with $K_{AB}\{N_2 – 1, N_3\}$ to prove he's Bob
    v. Alice replies with $K_{AB}\{N_3 – 1\}$ to prove she's Alice
b. Reflection Attack (in ECB mode)
    i. Trudy sends ticket | $K_{AB}(N_2)$ to Bob in ECB mode
    ii. Bob replies $K_{AB}\{N_2 – 1\}$ | $K_{AB}\{N_3\}$ (which only works in ECB mode)
    iii. Trudy needs to reply with $K_{AB}\{N_3 – 1\}$
    iv. So start a new connection.
        1. Trudy sends the ticket $K_{AB}\{N_3\}$ to Bob
        2. Bob replies with $K_{AB}\{N_3 – 1\}$ | $K_{AB}\{N_4\}$
        3. Now Trudy has the necessary reply for the original connection
    v. The solution: don't use ECB!
c. Limit Compromise
    i. Trudy steals Alice's key, then Alice changes it.
    ii. Gather an old ticket.
    iii. New algorithm
        1. A sends to B: I want to talk to you.
        2. B replies with $K_B\{N_B\}$ where maybe N_B is a timestamp
        3. A sends to the KDC: I want Bob, N_1, $K_B\{N_B\}$
        4. KDC replies to A: $K_A\{N_1$, "Alice", K_{AB}, $K_B\{K_{AB}$, "Alice", $N_B\}$
        5. Then do the same challenge/response as before

IX. Kerberos V4
a. Authentications using secret key cryptosystems
b. Tickets
    i. KDC sends $K_A\{K_{AB}\}$, $K_B\{K_{AB}$, Alice$\}$ ← the ticket
    ii. Expires in 21 hours
    iii. Every time you login, generate a new session key S_A
    iv. Gives a Ticket-Granting Ticket: $K_{KDC}\{S_A$, network info, expiration, …$\}$ where K_{KDC} is a key known only to the KDC itself.
    v. The workstation remembers S_A, and the TGT (ticket-granting ticket)
    vi. The workstation forgets the password
c. Configuration
    i. The KDC uses a master key to encrypt the password database and TGTs
    ii. Humans remember passwords; computers remember the key
d. Logging In
    i. Send the username, get credentials
    ii. KDC sends $K_A\{S_A$, TGT$\}$
    iii. Use the password to decrypt $K_A\{…\}$, then forget it.
    iv. Now you have S_A and a TGT to communicate with the KDC.
    v. Use S_A to generate new tickets when they expire.
e. Communicating with a Remote Node
    i. Send a message to the KDC that you want to talk to Bob
    ii. KDC generates K_{AB}, pulls S_A from the TGT.
    iii. Generates a ticket for Bob: $K_B\{$Alice, $K_{AB}\}$
    iv. The whole thing is encrypted with S_A to send to Alice.
    v. Alice can extract K_{AB} and the ticket for Bob
    vi. Use authenticator: K_{AB}(timestamp), allow for maybe five minute skew for clock synchronization

vii. Tickets in V4 contain the network address too so you can't just eavesdrop on an authentication and then login elsewhere if you're being malicious

f. A Problem
   i. If the KDC is down, nothing works
   ii. Want to replicate the KDC to avoid failure
   iii. Have a master KDC that accepts changes and replicated versions just provide services
   iv. Can exchange the master database in the clear, protected by a secure hash
g. Realms
   i. Multiple distinct KDCs
   ii. Can't create an arbitrary-length chain of KDCs.
   iii. Can talk to α and everybody in that realm.
   iv. If α can talk to β then you can talk to anybody in β's realm too.
   v. That's all though – no further.
h. A New Mode of Operation
   i. Plaintext Cipher Block Chaining (PCBC)
   ii. $C_{n+1} = E(m_{n+1} \oplus m_n \oplus c_n)$
   iii. If you corrupt c_i you'll ruin all c_j after that.
   iv. Put some recognizable string at the end, so you could detect tampering
   v. But: You can still swap two adjacent blocks