Password Pattern Studies

Klein - 1990
~ 20% used common words
  3% used their account name
~ 20% were from 6-8 characters long
Using resources at that time, 20% could be brute forced in a week

Spafford - 1992
20% were all lower case letter
  average length was still 7 characters

British bank - 2002
50% used family/pet names
~ 18% used famous people

Current studies based on recent password compromises
still finds a large percentage of pure lower case,
  pure numbers or lower case + numbers
most common lengths still 6-9 characters

Common patterns:
  lower case followed by numbers
  first letter capitalized, then lower case, then numbers
"1337" (leet) letter/number substitutions
  0/0 → Ø
  2/5 → 1
  ...

This drives the work done by mass-scale password crackers
Do multiple passes over user database
1) Dictionary attack
2) Append numbers to dictionary words
3) Capitalization permutations
4) Cap + numbers (combine 2 & 3)
5) Other modifications to dictionary (prepend numbers, append random, etc.)
6) Combine dictionary words
7) Random sequences up to n characters
8) Pure brute force

Encryption & Secure Authentication
One-way Hash Algorithms

$h(x)$ : hash function working on input $x$
$H$ : all possible hashes
$X$ : all possible input

$h(x) : x \rightarrow H$

Usually size of $X$ domain is larger than size of $H$ domain, so some overlap occurs ("collision")

General Principles of a Good Algorithm
1) Transforming $x$ to hash is fast
2) Transforming hash back to $x$ is computationally infeasible (must brute force)
3) Given $x$, it is computationally infeasible to find $x'$ such that $h(x) = h(x')$
4) Small changes to $x$ should result in random changes to $h(x)$

So if you have a hash & want $x$, you have to do trial & error attacks to find $x$
- generate a bunch of guesses & hash them until a match is found
- match is either original $x$ or a collision for $x$

"Fast" hash algorithms calculate $h(x)$ directly w/out any delay

"Slow" hash algorithms intentionally slow down calculating $h(x)$, such as doing multiple rounds of hashing

e.g. $\text{hash} = h(h(h(h(h(h(h(h(h(h(h(x)))))))))))$
"Salted" hashes - add a "salt" to x before calculating h(x)
- salt is just sequence of characters
- store the salt (plaintext) & the hash
  prevents pre-computing the hash
    e.g. take a dictionary
    for each word w, calculate h(w)
    store w & h(w) in database
    given a hash, see if there is a match
    in the database

How does password get transmitted to server?
1) Plaintext (HTTP, telnet, etc.)
2) Send the hash or encrypted hash (Kerberos)
   Client needs program that can calculate hash
   Kerberos exchange
   A: Alice wants to log in
   S: Authentication server

   A → S: Request to log in
   S → A: Authentication token (timestamped) encrypted
           with a key shared by A & S
   A → S: Hashed password encrypted with session key
           contained in authentication token

3) Send encrypted password
   - Issue: how to have a shared key between client & server
     that no one else knows

   Needham-Schroeder Key Exchange Protocol
   A: Alice (client)
   B: Bob (server)
   Na: nonce Alice chooses randomly
   Nb: nonce Bob chooses randomly
   KA: public key for Alice
   KB: public key for Bob

   A → B: Request to connect w/ KA
A → B: Request to connect w/ Ka

B → A: Response encrypted with Kg
contains part of polynomial function (based on)

Alice retrieves polynomial function piece & manipulates w/ Na to finish function

A → B: Response encrypted with Kg
contains enough data for B to finish polynomial function since he knows Nb

Bob & Alice can now calculate Ns but no one seeing just the partial function can calculate Ns

A → B: Sends password encrypted with Ns

Diffie–Hellman Key Exchange
Transmit some portions of equation such that both sides can calculate shared key but eavesdropper can’t

Method:
Sender chooses g & p
Sender chooses a random number a
Sender computes \( y = g^a \mod p \)
Sender transmits: \( y, g, p \)

Receiver chooses a random number b
Receiver computes \( y^b = g^b \mod p \)
Receiver computes \( y' = g^b \mod p \)
Receiver transmits: \( y', g^b, p \)

Receiver calculates \( (y')^a = g^{ab} \mod p \)

Shared key is \( g^{ab} \mod p \)