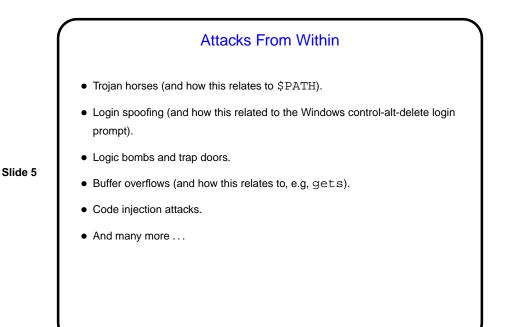
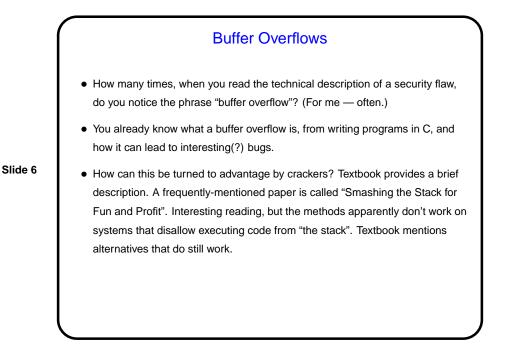
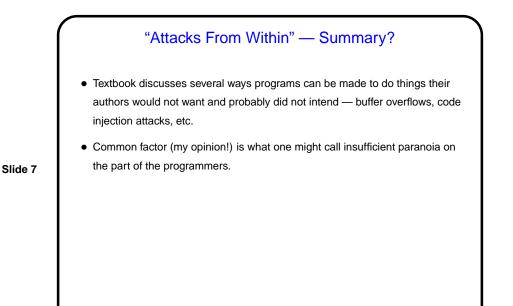


User Authentication
Based on "something the user knows" — e.g., passwords. Problems include where to store them, whether they can be guessed, whether they can be intercepted.
Based on "something the user has" — e.g., key or smart card. Problems include loss/theft, forgery.
Based on "something the user is" – biometrics. Problems include inaccuracy/spoofing.



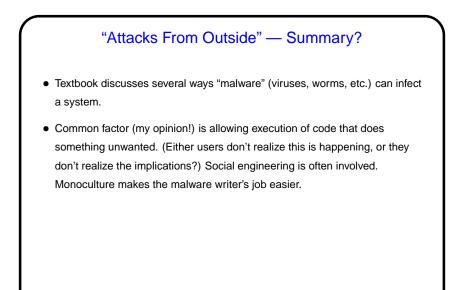




Attacks From Outside
Can categorize as viruses (programs that reproduce themselves when run), worms (self-replicating), spyware, etc. — similar ideas, though.
Many, many ways such code can get invoked — when legit programs are run, at boot time, when file is opened by some applications ("macro viruses"), etc.
Also many ways it can spread — once upon a time floppies were vector of choice, now networks or e-mail. Common factors:

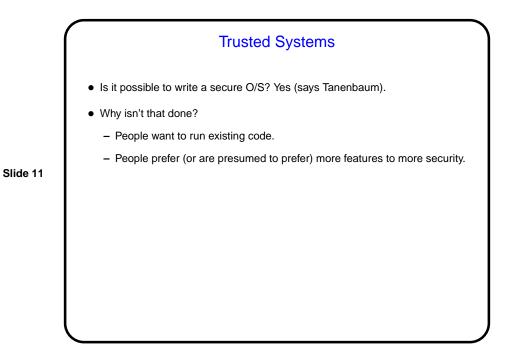
Executable content from untrustworthy source.
Human factors.

"Monoculture" makes it easier!
Virus scanners can check all executables for known viruses (exact or fuzzy matches), but hard/impossible to do this perfectly.
Better to try to avoid viruses — some nice advice in textbook.



Slide 10

## Safe Execution of "Mobile" Code Is there a way to safely execute code from possibly untrustworthy source? Maybe — approaches include sandboxing, interpretation, code signing. Example — Java's designed-in security: At source level, very type-safe — no way to use void\* pointers to access random memory. (Contrast with C!) When classes are loaded, "verifier" checks for potential security problems (not generated by normal compilers, but could be done by hand). At runtime, security manager controls what library routines are called — e.g., applets by default can't do file operations, many kinds of network access.



## Designing a Secure System "Security through obscurity" isn't very. Better to give too little access than too much — give programs/people as little as will work. Security can't be an add-on. "Keep it simple, stupid."

