reminder – implementation due 10/31
schedule demo with TAs 5-8pm

book does not cover secure coding

rule 1: do not use an unmanaged ("unsafe") language (C or C++)

rule 2: do not use a language where data can be executed (any if you're clever)

public enemy #1 – buffer overflow
only applies to unsafe languages

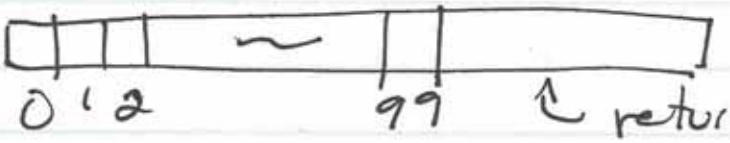
→ array accesses are unchecked because languages were designed when bounds checking was considered expensive

buffer overflow writes past the end of a buffer (usually an array of characters = string), overwriting whatever else was there

can cause core dump
can corrupt data structures

if the array is allocated on the program stack, then can overwrite return pointer of stack frame to jump to some memory address where "bad" code is inserted

→ explain stack vs. heap memory



char buffer[100];

example "bad" code: exec("/bin/sh");

Why can't we just close this loophole?

→ rewrite every C/C++ compiler to check array bounds
→ add array band checking code to every C/C++ program

another problem:

do not roll your own crypto
~~use a tried~~
use some standard API/library
(altho bad guys are always working on new attacks for standard tools)

let's say we're coding in a managed language & we're using an authentication package (and the implementations of the language virtual machine & the auth package do not themselves have ~~exploit~~ bugs that can be exploited)
→ still lots of other security threats

denial of service (DoS) attacks
malicious ~~use~~ user prevents your application from serving its legitimate users

one approach: force application crash
→ make sure all exceptions are handled in a way that enables the application to continue

CPU starvation
force application to take "a long time" processing user input - ~~bad input~~
→ put a time bound on processing each request, reject after bound, remember the bad user input immediately reject next time
(problem with zombienet attack where numerous apparently independent users send that same bad input)

example: application converts all double '/' inputs to single '/'
send input with n '/'s —
takes n^2 time

resource starvation

force application to consume too many resources
- memory
- database connections
- open files
- etc

do not allocate expensive resources until you know it's a legit user

even for legit users impose quotas (user may be hacked)

network bandwidth

do not reply to bad inputs
just quietly ~~reject~~ ignore

always run with "least privilege"
both application & OS level -
even if user IS an administrator
need to do something special to escalate privilege

default to deny privileges

base access on permissions not exclusion

NEVER trust user input even after authenticated

always validate all external inputs

never pass user input directly to a shell, interpreter, SQL query, etc.

always select components for query (SQL injection is major source of attacks)

→ different from input validation since component does not know it's external input

NEVER do security checks client side since modified client can approve anything

- always check at server that you control

for web applications, do not store sensitive data in cookies, hidden fields, or anything else that is sent to client

later used with new

sanitize all user inputs to HTTP POST - cross site scripting