robustness: the ability to handle or repel attacks

C Library
- Contains dangerous and safe APIs
  - Relatively old, invented in the 70s
  - Didn’t know back then what was so dangerous
- Examples
  - Dangerous: `gets(char *buf)`
    - Reads line from `stdin` (string terminated by `'
'`)
    - Return input buffer or NULL
  - Safe: `fgets(char *buf, int size, FILE *f)`
    - Includes a `size` argument
    - Reads line from `stdin`
      - Terminating at `'
'` or size (whichever first)
    - Returns input buffer

What makes a function dangerous?
- A dangerous function doesn’t give the user any method to prevent errors
- `gets`
  - Buffer is fixed size at particular address
  - `gets` returns data of user-specified length into buffer
  - Not dynamically-allocating space, so receiving code can’t change the size of it’s buffer
    - If you’re working with data defined by the user, and you can’t specify the size of the buffer, it’s a serious problem.
- `fgets`
  - Allows us to provide the size of the buffer as an argument
  - Safe

Stack smashing example
- We have some code
  - Allocates buffer with fixed size of 8204
  - Reads data into buffer with `gets`
- Attack
  - Provide input with size 8204, no newline
  - Then, provide address of malicious code to jump to
  - `gets` overwrites real address of code to jump to
  - Program jumps to our malicious code

Buffer overflow attack took down the entire internet
- Robert Morris (Kohler’s PhD advisor)
- http://en.wikipedia.org/wiki/Morris_worm

Fixing stack smashing
- Use safe APIs
- Can we automatically detect stack smashing? Yes!
gcc can detect stack smashing

- Option -fstack-detector
- Method __stack_chk_fail

On function entry
- Generate unpredictable number SM
  - Appears to be generated once per execution
- Store on stack
- Store copy in hidden storage (unpredictable location)

On function exit
- Load SM from hidden storage
- Compare with value on stack
- Fail if different
  - `mov 0x201c(%esp),%edx`
  - `xor %gs:0x14, %edx`
  - `je 0x8048f6d <main+141>`
- %gs is the hidden address (think of it as a global variable)
- Essentially placing “canary value” onto stack

Disadvantages
- Expensive operation
  - Creating a random number is expensive
  - However, compiler can identify functions which need stack
- Doesn’t prevent crashing the program
  - But, turns arbitrary execution failure into crash failure
  - Crash failure: bring down Amazon.com
  - Execution failure: TJ Maxx attack, steal credit cards
  - Harder to steal money with crash failure
    - Except through blackmail
    - Kohler seems to know a lot about this...
So...

Stack protection,
- Turns arbitrary execution failures into crash failures, which are harder to steal money.
  But they can, through blackmail “we’re gonna take you offline unless you give us money”

- Fixing stack smashing: an alternative method
  - On function entry, copy return address to hidden storage
  - On function exit, compare return address to hidden storage
  - Problem scenario
    - Three functions: f calls g calls h
    - The attacker could overwrite the variables, overwrite h’s ret address with the same value, overwrite g’s local variables, and put the attack in g’s return address. So it passes the stack protection of f, and g is vulnerable. So the attack works.
smash02.c

- Runs with stack protection
- Yet this is a program that's attackable

- Animal class
  - string *name
  - function pointer *will_eat of type (int (*)(animal *, const char *))
  - Function with return type int and arguments animal * and char *
  - pointer *next

- Will eat
  - Frog will eat a fly
  - Squirrel will eat a nut or pizza
  - Lion will eat everything

- Animals are stored in a linked list.
  - First defined is a squirrel, then a lion, and then a frog
- So what's the will_eat value set to?
  - Address that has static storage duration
  - We can take addresses of functions and stick them into data structures
  - Address of function that gets called is determined at runtime

- Linked list is search for the animal that is inputted, then it looks for the victim.
- Then if (a->will_eat(buf)) we take that function pointer and evoke it. The value of
  that function depends on the values that were put into the program at run time.

- call 0x4(%ebx)
- * means indirect call. 4 off of ebx is will_eat
There’s nothing in the program that prints bomb, but it does.

Where can we jump? We can jump to memory. What code is in memory?

The input is:
- lion
- `echo 'BOOM!!!'; sleep 5^@yyyyyyyyyyyy.....yyyyy0<A2>^D^H`

So what is <A2>^D^H? It looks like a return address.

SYSTEM executes a shell command
- This is not a stack smashing attack
- The buffer is located on the heap, it's dynamic
- The animal structures are placed after the buffer in memory
- The lion is the second animal. Each animal has a will_eat function ptr, and THIS is the value we're attacking, a dynamic function ptr stored in an object, not a return address.
- So this attack overwrites the buffer, one animal, and puts the address of the system function at a->will_eat. What's passed to a->will_eat? BUF. So if we put echo BOOM in buf, then we get the program to attack itself!
- Called a return-to-libc attack

---

Next Lecture
- `while (1);`
- The worst attack in the world
- Should bring any computer to a halt
- Small operating system
  - One process executing an infinite loop blocks all others
- Why this doesn't actually stop modern computers is super complicated