

# CSE380 - Operating Systems

Notes for Lecture 20 - 11/29/2007

© Matt Blaze

# RPC basics

- Basic idea: Split functions in two halves:
  - one or more *clients* may call the function
  - *server* actually runs it
- Problems:
  - how does the client identify the server computer?
  - how does the client identify what program it wants run on the server?
  - how does the client send the arguments?
  - how does the server send the return value?
  - what happens when messages get lost on network?

# rpcgen

- Nifty compiler for rpc programs
- Input is in “almost c”
  - C augmented by some syntax for RPC
  - conversion routines for XDR
  - input is in a “.x” file
- Output is two .c files plus a .h file
  - one .c file for the client (with stubs)
    - compile & link this into your client
  - another for the server (with the sever code)
    - compile and like with a small main() for the server
- Just run the generated server and you can call it with your client

# The single most important thing

WHEN YOU'RE DONE, MAKE SURE  
YOU KILL YOUR SERVER!

(you don't want your server running while  
you're logged out)

# Yeah, but what do you have do to make RPC work?

- Write the actual code for the remote service (in a .x file)
  - Select a service id for your service
    - this can't be the same as anything already running on the server machine
  - Use XDR format to translate complex data structures (including strings, structures, etc)
- Write a main program for the client
  - needs to call `clnt_create()` to identify server
- write a main program for the server
  - `rpcgen` can help with this (trivially simple)

# Important ideas from the Internet Protocols

- Dumb network, smart endpoints
  - “the end to end model”
  - network not designed for single application
  - endpoints do most of the work to make sense of packets
- Layering and encapsulation
  - link layer: local ethernet
  - network layer: routing packets over Internet
  - transport layer: reliable message streams
    - interpreted at host
  - application layer: mail, web, RPC, etc.
    - interpreted at host

# The link layer: Ethernet

- Everyone is connected to a big shared, common bus
- You can send *frames* on the bus
  - frames are just small messages
  - you wait for silence before sending
- Everyone has a unique (48 bit) address
  - frames have a sender and a receiver address
- You listen to all frames that go by
  - if a frame has your address, you process it
  - otherwise you politely ignore it

# Encapsulating Network Packets in Ethernet Frames

- Frames can contain IP *Datagrams*
  - An IP Datagram is a short message that also has a sender and receiver *network* address
- Some hosts are connected to more than one network
- When you get an Ethernet frame for your address, look at the packet inside
  - if the receiver network address is for you, process it locally
  - otherwise, pass it on on the appropriate network

# Hosts connected to $>1$ network are called *routers*

- They have to know how to get network datagrams closer to their destinations
  - Otherwise they're the same as other kinds of hosts
- If you want to send a datagram to a host on a different network, you send it to a router

OK, on to the next topic...

Security for Operating Systems  
(and networks)

# Security problems

- Securing Computers
  - how do we keep users on the same machine from interfering with one another?
  - how do we allow use of only parts of a machine?
- Securing Data
  - how do we protect data from eavesdropping or from being altered?
- Securing Networks
  - how do we protect data in transit?
  - can we protect network services?

# Let's do "Securing Data" first

- Why do this first?
  - basic tools for securing information can form the foundation for other security mechanisms (including those to secure the OS and the network)
  - plus it's something we may actually know how to do...
- What's the problem?
  - bits are just bits - can be copied, altered, etc.
  - computers, media, networks are inherently insecure
    - so bits stored on them are also insecure
- What's the solution?
  - cryptography (well, part of the solution anyway)

# *Media security vs. Message security*

- Modern systems tend to have *less* inherent security than older systems
  - more complexity == more bugs
  - more services that interact
  - more connectivity
- Internet exacerbates this
  - security is not a service of the network
- Need a way to send secure messages over insecure media
  - this is what cryptography does

# Cryptography

- Originally narrowly-defined
  - “secret writing”
  - message confidentiality (encryption)
- Now understood broadly, including
  - confidentiality (symmetric & asymmetric encryption)
  - identification (proof of secret knowledge)
  - message integrity (MAC codes)
  - signatures (digital signatures)
- Also includes surprising stuff
  - digital cash, voting, anonymity, and more

# Two observations about cryptography

- 1: Things that seem obviously easy turn out to be impossible
  - we have no proofs of security for almost anything
  - many cipher functions turn out to be broken
- 2: Things that seem obviously impossible turn out to be (relatively) easy
  - digital signatures, digital cash, public key crypto, mental poker
- Therefore: almost anything you think of will turn out to be wrong
  - so use standard, well evaluated solutions

# Symmetric Encryption

## (“secret key” cryptography)

- The most basic kind - what you'd think of first
  - monoalphabetic substitution is an example
    - A=X, B=M, C=R, D=A, etc.
- Basic idea: two functions, *encrypt* & *decrypt*
  - each take two arguments:  $\{text, key\}$ 
    - unencrypted called *plaintext*, encrypted called *ciphertext*
  - $C = \text{encrypt}(P, \text{key})$ ,  $P = \text{decrypt}(C, \text{key})$
  - hard to find P given C but not key
  - hard to find key even given  $\{P, C\}$
  - key shared by sender and receiver, kept secret
- Surprisingly hard to design a secure and efficient encryption function

# Symmetric Encryption

- Two kinds of encryption functions
  - block ciphers & stream ciphers
- Block ciphers encrypt *blocks* of bits
  - blocks usually 64-256 bits long
  - key defines a plaintext:ciphertext mapping
- Stream ciphers create *noise*
  - cipher uses key to generate a *keystream* of bits
  - encryption adds keystream, decryption subtracts it

# Using block ciphers: electronic code book mode

- Sender & receiver agree on secret key
  - needs to be unpredictable, random, secret
- Sender:
  - break msg into  $n$   $b$ -bit blocks  $P = \{P_1, P_2 \dots P_n\}$
  - calculate  $C_i = \text{encrypt}(P_i, \text{key})$  for all  $i < n$
  - sender sends  $C$  to receiver
- Receiver:
  - receive  $C$  from network
  - calculate  $P_i = \text{decrypt}(C_i, \text{key})$  for all  $i$
- Anyone can look at  $C$ , but that's OK

# Using block ciphers

- Mode as described works, but has limitations
  - what if message isn't a multiple of  $n$  bits?
  - a given  $P$  always encrypts to same  $C$  with the same key
    - so messages with repeated blocks leak information about their structure
- One solution: more complex *modes of operation* that *chain* successive blocks together
  - example: *cipher block chaining*
    - $C(i) = \text{encrypt}(P(i)+C(i-1), \text{key})$
    - $P(i) = \text{decrypt}(C(i), \text{key}) - C(i-1)$
    - usually use exclusive-or (why?)

# Stream ciphers

- Basic idea: functions that generate pseudorandom, cryptographically secure *streams* of bits based on key
  - hard to derive key given stream
  - hard to predict next bit given stream
- Sender and receiver share key and use it to generate *keystream*
  - sender:  $C_i = P_i + \text{stream}(\text{key}, i)$
  - receiver:  $P_i = C_i - \text{stream}(\text{key}, i)$
  - usually use exclusive-or (why?)

# Potential problems with secret key encryption

- Getting a secure cipher function
  - hard to design (anything you invent will be either insecure or inefficient or both)
  - Standard: Advanced Encryption Standard (AES)
- Agreeing on a key (and keeping it secret)
  - sender and receiver have to select a key together in secret and then keep it secret but available for use when they need it
- Building a *protocol*
  - just having the cipher isn't the same as having a secure *system*

# Public key cryptography to the rescue

- This is interesting: solves a seemingly impossible problem
  - first proposed in 1976 (Diffie-Hellman)
- General public key crypto allows there to be two keys, one for encryption and one for decryption
  - can't figure out decryption key from encryption key
  - so you can *publish* the encryption key
  - alice can just tell bob the key he should use to encrypt messages for her
- more on that later, but first, key agreement

# Diffie-Hellman key agreement

- First public key cryptosystem
  - Allows Alice and Bob to *agree* on a secret key
    - encrypt the actual message w/ secret key crypto
- Alice picks a random, secret  $x$ 
  - calculates  $A = g^x \pmod{p}$ , sends  $A$  to Bob (in public)
- Bob picks a random, secret  $y$ 
  - calculates  $B = g^y \pmod{p}$ , sends  $B$  to Alice (in public)
- They can each calculate  $K = g^{xy} \pmod{p}$ 
  - just requires exponentiation
  - Alice can calculate  $B^x \pmod{p}$
- But you can't! (logarithms are hard in a finite field)

# Putting it together

- Does Diffie-Hellman actually work?
  - does it give Alice and Bob a shared secret?
  - is it secure?
- How does it make designing a protocol easier?
- Meet Eve, the evesdropper
  - what if she sits between Alice and Bob?
  - how can we fix this?