ECS 289M Lecture 23

May 24, 2006

Measuring Capacity

- Intuitively, difference between unmodulated, modulated channel
  - Normal uncertainty in channel is 8 bits
  - Attacker modulates channel to send information, reducing uncertainty to 5 bits
  - Covert channel capacity is 3 bits
    - Modulation in effect fixes those bits

Formally

- Inputs:
  - *A* input from Alice (sender)
  - *V* input from everyone else
  - *X* output of channel
- Capacity measures uncertainty in *X* given *A*
- In other terms: maximize

$$I(A; X) = H(X) - H(X \mid A)$$

  with respect to *A*

Example (continued)

- If A, V independent, p=p(A=0), q=p(V=0):
  - p(A=0,V=0) = pq
  - p(A=1,V=0) = (1–p)q
  - p(A=0,V=1) = p(1–q)
  - p(A=1,V=1) = (1–p)(1–q)
- So
  - p(X=0) = p(A=0,V=0)+p(A=1,V=1)
    = pq + (1–p)(1–q)
  - p(X=1) = p(A=0,V=1)+p(A=1,V=0)
    = (1–p)q + p(1–q)

More Example

- Also:
  - $p(X=0|A=0) = q$
  - $p(X=0|A=1) = 1-q$
  - $p(X=1|A=0) = 1-q$
  - $p(X=1|A=1) = q$
- So you can compute:
  - $H(X) = -[(1-p)q + p(1-q)] \lg [(1-p)q + p(1-q)]$
  - $H(X|A) = -q \lg q - (1-q) \lg (1-q)$
  - $I(A;X) = H(X)-H(X|A)$

$I(A;X)$

$$I(A; X) = -[pq + (1-p)(1-q)] \lg [pq + (1-p)(1-q)] - [(1-p)q + p(1-q)] \lg [(1-p)q + p(1-q)] + q \lg q + (1-q) \lg (1-q)$$

- Maximum when p = 0.5; then

  $I(A;X) = 1 + q \lg q + (1-q) \lg (1-q) = 1-H(V)$
- So, if V constant, $q = 0$, and $I(A;X) = 1$
- Also, if $q = p = 0.5$, $I(A;X) = 0$

Analyzing Capacity

- Assume a noisy channel
- Examine covert channel in MLS database that uses replication to ensure availability
  - 2-phase commit protocol ensures atomicity
  - *Coordinator* process manages global execution
  - *Participant* processes do everything else

How It Works

- Coordinator sends message to each participant asking whether to abort or commit transaction
  - If any says "abort", coordinator stops
- Coordinator gathers replies
  - If all say "commit", sends commit messages back to participants
  - If any says "abort", sends abort messages back to participants
  - Each participant that sent commit waits for reply; on receipt, acts accordingly

Exceptions

- Protocol times out, causing party to act as if transaction aborted, when:
  - Coordinator doesn't receive reply from participant
  - Participant who sends a commit doesn't receive reply from coordinator

Covert Channel Here

- Two types of components
  - One at *Low* security level, other at *High*
- Low component begins 2-phase commit
  - Both *High*, *Low* components must cooperate in the 2-phase commit protocol
- *High* sends information to *Low* by selectively aborting transactions
  - Can send abort messages
  - Can just not do anything

Note

- If transaction *always* succeeded except when *High* component sending information, channel not noisy
  – Capacity would be 1 bit per trial
  – But channel noisy as transactions may abort for reasons *other* than the sending of information

Analysis

- *X* random variable: what *High* user wants to send
  – Assume abort is 1, commit is 0
  – $p = p(X=0)$ probability *High* sends 0
- *A* random variable: what *Low* receives
  – For noiseless channel $X = A$
- $n+2$ users
  – Sender, receiver, *n* others
  – *q* probability of transaction aborting at any of these *n* users

Basic Probabilities

- Probabilities of receiving given sending
  - $p(A=0|X=0) = (1-q)^n$
  - $p(A=1|X=0) = 1-(1-q)^n$
  - $p(A=0|X=1) = 0$
  - $p(A=1|X=1) = 1$
- So probabilities of receiving values:
  - $p(A=0) = p(1-q)^n$
  - $p(A=1) = 1-p(1-q)^n$

More Probabilities

- Given sending, what is receiving?
  - $p(X=0|A=0) = 1$
  - $p(X=1|A=0) = 0$
  - $p(X=0|A=1) = p[1-(1-q)^n] / [1-p(1-q)^n]$
  - $p(X=1|A=1) = (1-p) / [1-p(1-q)^n]$

Entropies

- $H(X) = -p \lg p - (1-p) \lg (1-p)$
- $H(X|A) = -p[1-(1-q)^n] \lg p$
  $- p[1-(1-q)^n] \lg [1-(1-q)^n]$
  $+ [1-p(1-q)^n] \lg [1-p(1-q)^n]$
  $- (1-p) \lg (1-p)$
- $I(A;X) = -p(1-q)^n \lg p$
  $+ p[1-(1-q)^n] \lg [1-(1-q)^n]$
  $- [1-p(1-q)^n] \lg [1-p(1-q)^n]$

Capacity

- Maximize this with respect to p (probability that *High* sends 0)
  - Notation: $m = (1-q)^n$, $M = (1-m)^{(1-m)}$
  - Maximum when $p = M / (Mm+1)$
- Capacity is:

$$I(A;X) = \frac{Mm \lg p + M(1-m) \lg (1-m) + \lg (Mm+1)}{(Mm+1)}$$

Mitigation of Covert Channels

- Problem: these work by varying use of shared resources
- One solution
  - Require processes to say what resources they need before running
  - Provide access to them in a way that no other process can access them
- Cumbersome
  - Includes running (CPU covert channel)
  - Resources stay allocated for lifetime of process

Alternate Approach

- Obscure amount of resources being used
  - Receiver cannot distinguish between what the sender is using and what is added
- How? Two ways:
  - Devote uniform resources to each process
  - Inject randomness into allocation, use of resources

Uniformity

- Variation of isolation
  - Process can't tell if second process using resource
- Example: KVM/370 covert channel via CPU usage
  - Give each VM a time slice of fixed duration
  - Do not allow VM to surrender its CPU time
    - Can no longer send 0 or 1 by modulating CPU usage

Randomness

- Make noise dominate channel
  - Does not close it, but makes it useless
- Example: MLS database
  - Probability of transaction being aborted by user other than sender, receiver approaches 1
    - $q \to 1$
  - $I(A; X) \to 0$
  - How to do this: resolve conflicts by aborting increases q, or have participants abort transactions randomly

Problem: Loss of Efficiency

- Fixed allocation, constraining use
  – Wastes resources
- Increasing probability of aborts
  – Some transactions that will normally commit now fail, requiring more retries
- Policy: is the inefficiency preferable to the covert channel?

Example

- Goal: limit covert timing channels on VAX/VMM
- "Fuzzy time" reduces accuracy of system clocks by generating random clock ticks
  – Random interrupts take any desired distribution
  – System clock updates only after each timer interrupt
  – Kernel rounds time to nearest 0.1 sec before giving it to VM
    - Means it cannot be more accurate than timing of interrupts

Example

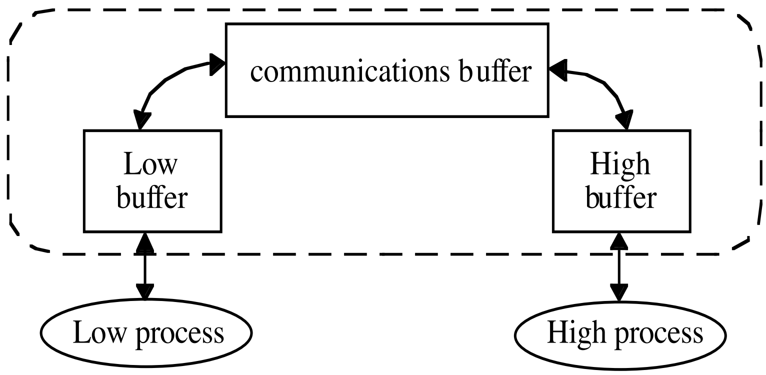
- I/O operations have random delays
- Kernel distinguishes 2 kinds of time:
  - *Event time* (when I/O event occurs)
  - *Notification time* (when VM told I/O event occurred)
    - Random delay between these prevents VM from figuring out when event actually occurred)
    - Delay can be randomly distributed as desired (in security kernel, it's 1–19ms)
  - Added enough noise to make covert timing channels hard to exploit

Improvement

- Modify scheduler to run processes in increasing order of security level
  - Now we're worried about "reads up", so …
- Countermeasures needed only when transition from *dominating* VM to *dominated* VM
  - Add random intervals between quanta for these transitions

The Pump

- Tool for controlling communications path between *High* and *Low*



Details

- Communications buffer of length *n*
  – Means it can hold up to *n* messages
- Messages numbered
- Pump ACKs each message as it is moved from *High* (*Low*) buffer to communications buffer
- If pump crashes, communications buffer preserves messages
  – Processes using pump can recover from crash

Covert Channel

- Low fills communications buffer
  - Send messages to pump until no ACK
  - If *High* wants to send 1, it accepts 1 message from pump; if *High* wants to send 0, it does not
  - If *Low* gets ACK, message moved from *Low* buffer to communications buffer ⇒ *High* sent 1
  - If *Low* doesn't get ACK, no message moved ⇒ *High* sent 0
- Meaning: if *High* can control rate at which pump passes messages to it, a covert timing channel

Performance vs. Capacity

- Assume *Low* process, pump can process messages more quickly than *High* process
- L_i random variable: time from *Low* sending message to pump to *Low* receiving ACK
- H_i random variable: average time for *High* to ACK each of last n messages

Case1: $E(L_i) > H_i$

- *High* can process messages more quickly than *Low* can get ACKs
- Contradicts above assumption
  - Pump must be delaying ACKs
  - *Low* waits for ACK whether or not communications buffer is full
- Covert channel closed
- Not optimal
  - Process may wait to send message even when there is room

Case 2: $E(L_i) < H_i$

- *Low* sending messages faster than *High* can remove them
- Covert channel open
- Optimal performance

Case 3: $E(L_i) = H_i$

- Pump, processes handle messages at same rate
- Covert channel open
  – Bandwidth decreased from optimal case (can't send messages over covert channel as fast)
- Performance not optimal

Adding Noise

- Shown: adding noise to approximate case 3
  – Covert channel capacity reduced to 1/*nr* where *r* time from *Low* sending message to pump to *Low* receiving ACK when communications buffer not full
  – Conclusion: use of pump substantially reduces capacity of covert channel between *High*, *Low* processes when compared to direct connection

Trojan Horse

- Program with an *overt* purpose (known to user) and a *covert* purpose (unknown to user)
  - Often called a Trojan
  - Named by Dan Edwards in Anderson Report

Example

- Shell script on a UNIX system:

```
cp /bin/sh /tmp/.xyzzy
chmod u+s,o+x /tmp/.xyzzy
rm ./ls
ls $*
```

- Place in program called "ls" and trick someone into executing it
- You now have a setuid-to-*them* shell!

Example: NetBus

- Designed for Windows NT system
- Victim uploads and installs this
  - Usually disguised as a game program, or in one
- Acts as a server, accepting and executing commands for remote administrator
  - This includes intercepting keystrokes and mouse motions and sending them to attacker
  - Also allows attacker to upload, download files

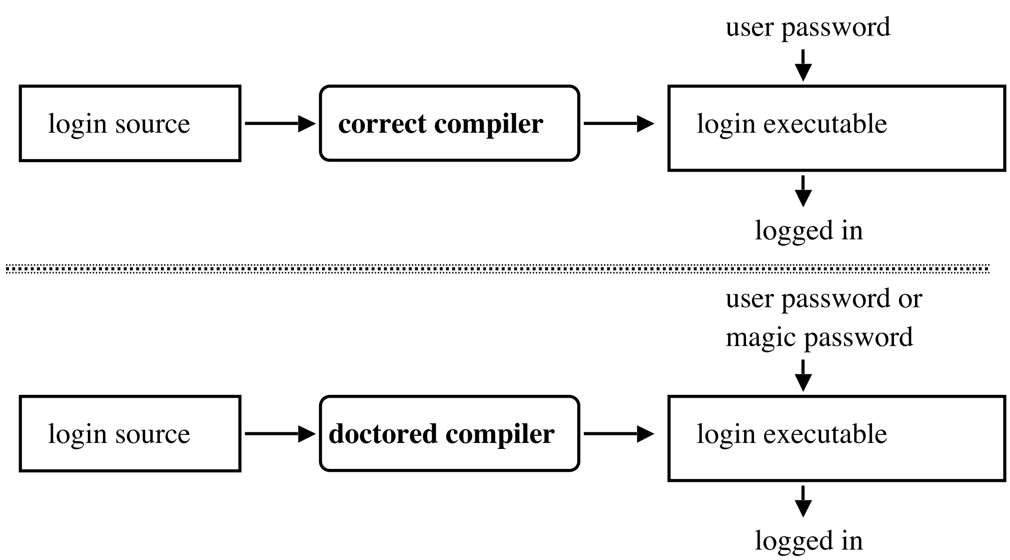
Replicating Trojan Horse

- Trojan horse that makes copies of itself
  - Also called *propagating Trojan horse*
  - Early version of *animal* game used this to delete copies of itself
- Hard to detect
  - 1976: Karger and Schell suggested modifying compiler to include Trojan horse that copied itself into specific programs including later version of the compiler
  - 1980s: Thompson implements this

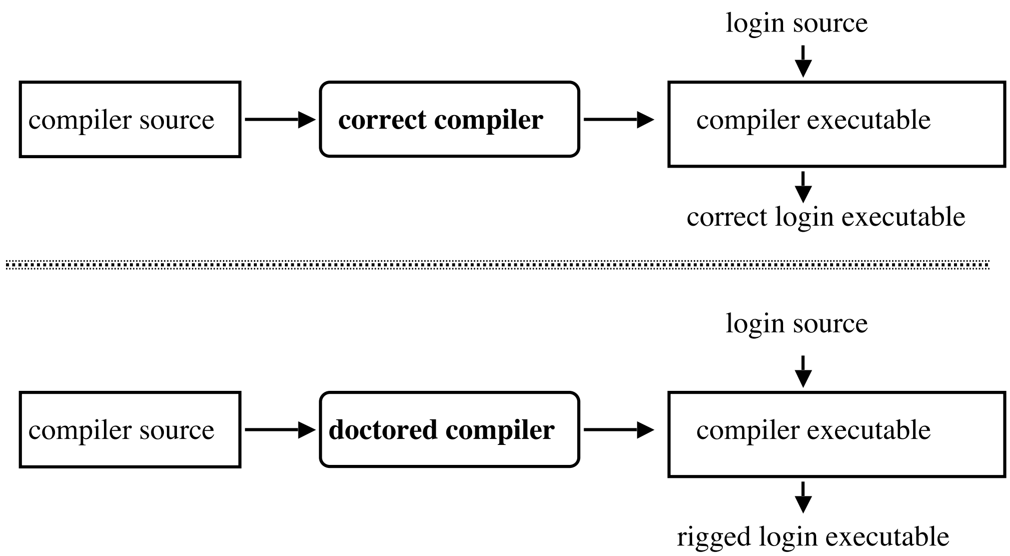
Thompson's Compiler

- Modify the compiler so that when it compiles *login* , *login* accepts the user's correct password or a fixed password (the same one for all users)
- Then modify the compiler again, so when it compiles a new version of the compiler, the extra code to do the first step is automatically inserted
- Recompile the compiler
- Delete the source containing the modification and put the undoctored source back

The Login Program



The Compiler



Comments

- Great pains taken to ensure second version of compiler never released
  - Finally deleted when a new compiler executable from a different system overwrote the doctored compiler
- The point: *no amount of source-level verification or scrutiny will protect you from using untrusted code*
  - Also: having source code helps, but does not ensure you're safe