Outline

- What is computer security?
  - Protecting against worms and viruses?
  - Making sure programs obey their specifications?
  - Still plenty of security problems even if these problems are solved...

Acknowledgments: Steve Zdancewic, Fred Schneider

What is security?

- Security: prevent bad things from happening
  - Confidential information leaked
  - Important information damaged
  - Critical services unavailable
  - Clients not paying for services
  - Money stolen
  - Improper access to physical resources
  - System used to violate law
  - Loss of value
  ... or at least make them less likely
- Versus an adversary!

Attack Sampler #1: Morris Worm

1988: Penetrated an estimated 5 to 10 percent of the 6,000 machines on the internet. Used a number of clever methods to gain access to a host.
  - brute force password guessing
  - bug in default sendmail configuration
  - X windows vulnerabilities, rlogin, etc.
  - buffer overrun in fingerd
Remarks:
  - System diversity helped to limit the spread.
  - “root kits” for cracking modern systems are easily available and largely use the same techniques.

2002: MS-SQL Slammer worm

- Jan. 25, 2002: SQL and MSDE servers on Internet turned into worm broadcasters
  - YABO
  - Spread to most vulnerable servers on the Internet in less than 10 min!
- Denial of Service attack
  - Affected databases unavailable
  - Full-bandwidth network load ⇒ widespread service outage
  - “Worst attack ever” – brought down many sites, not Internet
- Can’t rely on patching!
  - Infected SQL servers at Microsoft itself
  - Owners of most MSDE systems didn’t know they were running it… support for extensibility

Attack sampler #2: Love Bug, Melissa

- 1999: Two email based viruses that exploited:
  - a common mail client (MS Outlook)
  - trusting (i.e., uneducated) users
  - VB scripting extensions within messages to:
    - look up addresses in the contacts database
    - send a copy of the message to those contacts
- Melissa: hit an estimated 1.2 million machines.
- Love Bug: caused estimated $10B in damage.
- Remarks:
  - no passwords, crypto, or native code involved
Attack sampler #3: Hotmail

- 1999: All Hotmail email accounts fully accessible by anyone, without a password
- Just change username in an access URL (no programming required!)
- Selected other Hotmail headlines (1998-99)
  - Hotmail bug allows password theft
  - Hotmail bug pops up with JavaScript code
  - Malicious hacker steals Hotmail passwords
  - New security glitch for Hotmail
  - Hotmail bug fix not a cure-all

Attack sampler #4: Yorktown

- 1998: “Smart Ship” USS Yorktown suffers propulsion system failure, is towed into Norfolk Naval Base
- Cause: computer operator accidentally types a zero, causing divide by zero error that overflows database and crashes all consoles
- Problem fixed two days later

Attack sampler #5: insiders

- Average cost of an outsider penetration is $56,000; average insider attack cost a company $2.7 million (Computer Security Institute/FBI)
- 63 percent of the companies surveyed reported insider misuse of their organization’s computer systems. (WarRoom Research)
- Some attacks:
  - Backdoors
  - “Logic bombs”
  - Holding data hostage with encryption
  - Reprogramming cash flows
- Attacks may use legitimately held privileges!
- Many attacks (90%?) go unreported

Terminology

- Vulnerability: Weakness that can be exploited in a system
- Attack: Method for exploiting vulnerability
- Threat / Threat model: The power of the attacker (characterizes possible attacks)
  - E.g., attacker can act as an ordinary user, read any data on disk, and monitor all network traffic.
- Trusted Computing Base: Set of system components that are depended on for security
  - Usually includes hardware, OS, some software, …

Who are the attackers?

- Operator/user blunders.
- Hackers driven by intellectual challenge (or boredom).
- Insiders: employees or customers seeking revenge or gain
- Criminals seeking financial gain.
- Organized crime seeking gain or hiding criminal activities.
- Organized terrorist groups or nation states trying to influence national policy.
- Foreign agents seeking information for economic, political, or military purposes.
- Tactical countermeasures intended to disrupt military capability.
- Large organized terrorist groups or nation-states intent on overthrowing the US government.

What are the vulnerabilities?

- Poorly chosen passwords
- Software bugs
  - unchecked array access (buffer overflow attacks)
- Automatically running active content: macros, scripts, Java programs
- Open ports: telnet, mail
- Incorrect configuration
  - file permissions
  - administrative privileges
- Untrained users/system administrators
- Trap doors (intentional security holes)
- Unencrypted communication
- Limited Resources (i.e. TCP connections)
What are the attacks?

- Password Crackers
- Viruses:
  - LoveYou (VBscript virus), Melissa (Word macro virus)
- Worms:
  - Code Red: Port 80 (HTTP), Buffer overflow in IIS (Internet/Indexing Service)
- Trojan Horses
- Root kits, Back Orifice, SATAN
- Social Engineering:
  - “Hi, this is Joe from systems, I need your password to do an upgrade”
- Packet sniffers: Ethereal
- Denial of service: TCP SYN packet floods

Social engineering attacks

- Email this to a friend
- Print this version.

Passwords revealed by sweet deal

More than 70% of people would reveal their computer password in exchange for a bar of chocolate, a survey has found.

It also showed that 34% of respondents volunteered their password when asked without even needing to be bribed.

A second survey found that 79% of people unwittingly gave away information that could be used to steal their identity when questioned.

Security firms predict that lax security practices will fuel a British boom in online identity theft.

Security vs. fault tolerance

- Attacks vs. faults
- Reliability community often assumes benign, random faults
  - Failstop failures = system halts
  - Byzantine failure = system behaves arbitrarily badly (under control of adversary)
- Attackers go for the weakest link!
  - It doesn’t help to have a $1000 lock on your door if the window is open.

Assumptions and abstraction

- Arguments for security always rest on assumptions:
  - “the attacker does not have physical access to the hardware”
  - “the code of the program cannot be modified during execution”
- Assumptions are vulnerabilities
  - Sometimes known, sometimes not
- Assumptions arise from abstraction
  - security analysis only tractable on a simplification (abstraction) of actual system
  - Abstraction hides details (assumption: unimportant)

Risk management

- Cost benefit: high security may be more expensive than benefits obtained
  - Security measures interfere with intended use
  - functionality
  - efficiency
  - cost
  - Preventing problems may be infeasible, unnecessary; deterrence may be sufficient
    - Remove the incentive to attack
    - Make it easier to attack someone else
    - Make it too costly to attack

When to enforce security

Possible times to respond to security violations:
- Before execution: analyze, reject, rewrite
- During execution: monitor, log, halt, change
- After execution: roll back, restore, audit, sue, call police
**Policy vs. mechanism**

- What is being protected (and from what) vs.
- How it is being protected
  (access control, cryptography, …)

- Want:
  - To know what we need to be protected from
  - Mechanisms that can implement many policies

**What is being protected?**

- Something with value
- Information with (usually indirect) impact on real world
- Different kinds of protection are needed for different information: ensure different security properties
  - Confidentiality
  - Integrity
  - Availability

**Properties: Integrity**

- No improper modification of data

- E.g., account balance is updated only by authorized transactions, only you can change your password
- Integrity of security mechanisms is crucial
- Enforcement: access control, digital signatures,…

**Properties: Confidentiality**

- Protect information from improper release

- Limit knowledge of data or actions
- E.g., D-Day attack date, contract bids
- Also: secrecy
- Enforcement: access control, encryption,…
- Hard to enforce after the fact…

**Properties: Privacy, anonymity**

- Related to confidentiality
- Privacy: prevent misuse of personal information
- Anonymity: prevent connection from being made between identity of actor and actions
  - Keep identity secret
  - Keep actions secret

**Properties: Availability**

- Easy way to ensure confidentiality, integrity: unplug computer

- Availability: system must respond to requests

Data
**Properties: Nonrepudiation**

- Ability to convince a third party that an event occurred (e.g., sales receipt)
- Needed for external enforcement mechanisms (e.g., police)
- Related to integrity

**Is security just correctness?**

- "System is secure" ≠ "System obeys specification"
- Specifications usually focus on functionality, not security
- Classic specification languages (e.g. Hoare logic) don’t talk about security properties
- Security is not preserved under refinement
  
  ```
  public ∈ Z looks secure
  public := secret isn't
  ```

**Safety properties**

- "Nothing bad ever happens" (at a particular moment in time)
- A property that can be enforced using only history of program
- Amenable to purely run time enforcement
- Examples:
  - access control (e.g. checking file permissions on file open)
  - memory safety (process does not read/write outside its own memory space)
  - type safety (data accessed in accordance with type)

**Liveness properties**

- "Something good eventually happens"
- Example: availability
  - "The email server will always respond to mail requests in less than one second"
- Violated by denial of service attacks
- Can’t enforce purely at run time – stopping the program violates the property!
- Tactic: restrict to a safety property
  - "web server will respond to page requests in less than 10 sec or report that it is overloaded."

**Security Property Landscape**

"System does exactly what it should—and no more"

- Privacy
- Digital rights
- Noninterference (confidentiality, integrity)

<table>
<thead>
<tr>
<th>Security Mechanisms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summer School on Software Security</td>
</tr>
<tr>
<td>June 2004</td>
</tr>
<tr>
<td>Andrew Myers</td>
</tr>
<tr>
<td>Cornell University</td>
</tr>
</tbody>
</table>
Topics

- Fundamental enforcement mechanisms
- Design principles for secure systems
- Operating system security mechanisms

Mechanisms: Authentication

- If system attempts to perform action X, should it be allowed? (e.g., read a file)
  - authentication + authorization
- **Authentication:** what principal p is system acting on behalf of? Is this an authentic request from p?
  - Passwords, biometrics, certificates...

Principals

- A principal is an identity; an abstraction of privileges
  - Process uid
  - E.g., a user (Bob), a group of users (Model airplane club), a role (Bob acting as president)

Mechanisms: Authorization

- **Authorization:** is principal p authorized to perform action A?
- Access control mediates actions by principals
- Example: file permissions (ACLs)

Mechanisms: Auditing

- For after the fact enforcement, need to know what happened: auditing
- Audit log records security relevant actions (who, what, when)
- Authorization + Authentication + Audit = “The gold (Au) standard”: classic systems security
- A fourth kind of mechanism: program analysis and verification
  - Needed for extensible systems and strong security properties… more later

Principle: Complete Mediation

- Common requirement: system must have ability to mediate all security relevant operations
  - Dangerous to assume op is not security-relevant...
  - Many places to mediate: hardware, compiler, ...
- Assumption: mediation mechanism cannot be compromised (TCB)
- Example: operating system calls
  - Kernel interface mediates access to files, memory pages, etc.
  - No other way to create/manipulate resources
  - One problem: covert timing channels
Principle: Minimize TCB
- Observation: Complex things are more likely not to work correctly
- Economy of Mechanism: Make trusted computing base as small and simple as possible.
  “Things should be made as simple as possible—but no simpler.” — A. Einstein
- Fewer errors in implementation, easier to convince someone that it’s correct
- Corollary: Failsafe Defaults
  - Access should be off by default, explicitly enabled

Principle: Least Privilege
- A principal should be given only those privileges needed to accomplish its tasks.
  - No more, no less.
- What is the minimal set of privileges?
- What is the granularity of privileges?
  - Separation of privileges (read vs. write access)
- How & when do the privileges change?
- Example violation: UNIX sendmail has root privilege

Principle: Open Design
- Success of mechanism should not depend on it being secret
  - “No security through obscurity”
  - The only secrets are cryptographic keys
  - Increased assurance if many critics.
- An age-old controversy:
  - Open design makes critics’ jobs easier, but also attackers’ job.
  - Analysis tends to concentrate on core functionality; vulnerabilities remain off the beaten path. (Ergo: small TCB)

Principle: Security is a Process
- Every system has vulnerabilities
  - Impossible to eliminate all of them
  - Goal: assurance
- Systems change over time
  - Security requirements change over time
  - Context of mechanisms changes over time
- Secure systems require maintenance
  - Check for defunct users
  - Update virus software
  - Patch security holes
  - Test firewalls

Conventional security mechanisms
- Access control, encryption, firewalls, memory protection, ...
- What are they?
- What are they good for?
- Where do they fall short?

Operating system security
- Program is black box
- Program talks to OS via mediating interface (system calls)
  - Multiplex hardware
  - Isolate processes from each other
  - Restrict access to persistent data (files)
- Language independent, simple
Weaknesses
- Treating the program as a black box
  - Not fine-grained enough to enforce desired properties
  - No help with validation
  - Internal behavior of program is important!

Reference Monitor
- Observes the execution of a program and halts the program if it’s going to violate the security policy
- Common Examples:
  - memory protection
  - access control checks
  - routers
  - firewalls
- Most current enforcement mechanisms are reference monitors

Access control
- A mechanism for controlling which actions are permitted
- Assumes a reference monitor
- Can enforce safety properties
- Local but not system wide enforcement of confidentiality and integrity

ACLs
- Access control list maps principals to their privileges
- Reference monitor checks relevant operations against ACL
- Works well if
  - Privileges have right granularity
  - System is not too complex

Capabilities
- Capability is an object that confers privileges to the possessor
- Important property: capabilities cannot be forged
- Different capability representations
  - Cryptographically strong pseudorandom number
  - Held by operating system ala file descriptors (Mach)
  - Object reference (Java)
- Advantage: allows privileges to be delegated even outside local system
  - Hard to keep capabilities from leaking out
  - Revoking capabilities can be difficult, expensive
  - E.g., X.509

Java: objects as capabilities
- Single Java VM may contain processes with different levels of privilege (e.g. different applets)
- Some objects are capabilities to perform security relevant operations:
  ```java
  FileReader f = new FileReader("/etc/passwd");
  // now use "f" to read password file
  ```
- Original 1.0 security model: use type safety, encapsulation to prevent untrusted applets from accessing capabilities in same VM
- Problem: tricky to prevent capabilities from leaking (downcasts, reflection, …)
**Mandatory access control**

- Ordinary access control only protects information at point of access
- Confidentiality: program may leak information after it reads
- Integrity: program may overwrite with data from untrustworthy sources

**Discretionary access control:**

- No control of propagation (at discretion of reader)

**Mandatory access control/multilevel security:**

- Attach security labels to data, processes

**MAC Problems**

- Read from a location with higher security label either:
  - Is rejected (no read-up / simple security property)
  - Raises the label of the process
- Write to a location with a lower security label either:
  - Is rejected (no write-down / ^-property)
  - Raises the label of the location

- No write-down is awkward
- Label creep makes data unusable
- Expensive
- Not used much!

**Cryptography (very briefly)**

- Can construct algorithms that compute functions $f$ such that $x$ cannot be recovered from $f(x)$
- Keys $k$ parameterize general algorithms ($E,D$)
- Shared-key cryptography: $E(k)$ is inverse of $D(k)$
  - $D(k, E(k, m)) = m$
  - Example: DES
  - Problem: distributing shared keys securely
- Public-key cryptography: $E(k_p)$ is inverse of $D(k_d)$, but cannot find $k_d$ even given $k_p$
  - $D(k_d, E(k_p, m)) = m = E(k_p, D(k_d, m))$
  - $k_p$ is public key, $k_d$ is corresponding secret key
  - Example: RSA
  - Problem: expensive
- Secure hashing: $m$ cannot be recovered from $H(m)$
  - Example: MD5

**Using cryptography**

- Encryption:
  - $E(k, m)$ keeps $m$ from those who do not have key $k$: protects confidentiality
  - $E(k, m)$ or $D(k, m)$ can convince that you have $k$
  - $E(k_p, m)$ keeps $m$ secret from those who do not have $k_p$ (and sender doesn’t need a secret)
  - Makes key distribution much easier
- Digital signatures:
  - $D(k_p, m)$ proves that message came from principal holding $k_p$
  - Anyone can check because $m = E(k_p, D(k_p, m))$
  - Provides authentication, integrity, nonrepudiation
  - Public keys stand for principals

**Intrusion detection?**

- Monitor behavior of programs and take remedial action if behavior is malicious or suspicious (anomaly detection)
  - Signal to operator, halt processes, roll back changes…
  - Can monitor at any level supporting mediation
- Inspired by biological systems
- Problems:
  - False alarms
  - Run-time overhead
  - Instability/autoimmune disease
  - Argument for higher assurance?
    - We do this anyway – but tools help!
Virus scanning?
• Scan for suspicious code
  – e.g., McAfee, Norton, etc.
  – based largely on a lexical signature.
  – the most effective commercial tool
  – but only works for things you’ve seen
    • Melissa spread in a matter of hours
  – virus kits make it easy to disguise a virus
    • “polymorphic” viruses
• Doesn’t help as much with worms
  (some network-packet scanning tools)

Distribution/partitioning
• Computation in general involves cooperation
  between mutually distrustful principals
• Securely place
  computation, data
  • User’s bank balance
  • User order history
  • Corporate partners
  • Browser
  • Spreadsheet
  • Javascript
  • Servlets
  • Web server
  • Other corporate info

Replication
• Can improve integrity at the expense of
  availability:
  • Pick first
• Can improve availability at the expense of
  integrity:
  • Pick first

Rollback/Undo
• Many systems (esp. databases) have a that
  records all changes made during a transaction
• Used to make transactions appear atomic
• Idea: use log to roll back changes

Replication
• Can improve both:
  • Quorum systems, etc.

Interposition
• Complete mediation: should be able to intercept
  security-relevant operations
• May not know what is security-relevant at design time
  – Systems evolve and are used in unexpected ways
• Need general mechanisms for extensible mediation
  – Virtual machine monitors (e.g., VMware)
  – Software virtual machines
  – Program transformation (sandboxing/SFI, inlined
    reference monitors)
• Problem: recognizable operations may be at wrong
  level of abstraction
Information Flow Security

Summer School on Software Security
June 2004
Andrew Myers
Cornell University

End-to-end security

- Near term problem: ensuring programs are memory safe, type safe so fine-grained access control policies can be enforced
- Long term problem: ensuring that complex (distributed) computing systems enforce end to end information security policies
  - Confidentiality
  - Integrity
  - Availability
- Confidentiality, integrity: end to end, security described by information flow policies

Information security: confidentiality

- Simple (access control) version:
  - Only authorized processes can read a file
  - But... when should a process be “authorized”?  
  - Encryption provides end-to-end confidentiality—if no computation
- End to end version:
  - Information should not be improperly released by a computation no matter how it is used
  - Requires tracking information flow

Information security: integrity

- Simple (access control) version:
  - Only authorized processes can write a file
  - But... when should a process be “authorized”?  
  - Digital signatures provide end to end integrity—if no computation
- End-to-end version:
  - Information should not be updated on the basis of less trustworthy information

Intentional vs. extensional security

- Access control is intentional: security requirements expressed in terms of program artifacts
  - Authority of processes and programs
  - File permissions
- Information flow is (ideally) extensional – regulates observable behavior of program rather than internals

Information channels

- End to end security requires controlling information channels (Lampson73)
- Storage channels: explicit information transmission (writes to sockets, files, variable assignments)
- Covert channels: transmit by mechanisms not intended for signaling information (system load, run time, locks)
- Timing channels: transmit information by when something happens (rather than what)
Implicit flows

- Covert storage channels arising from control flow. Example:
  ```java
  boolean b := <some secret>
  if (b) {
      x = true; f();
  }
  ```
- Creates information flow from b to x
- Run time check requires whole process labeled secret after branch

Multilevel security (MLS)

- Originally, computers, networks segregated by security class of information used
  - E.g., information could go from unclassified network to classified network but not vice versa
- Idea: build one system that can securely manipulate information of different classes
  - Multilevel secure: goal is end-to-end secrecy
  - Mandatory access control one possible
- One attempt: Multics/AIM ring model
  - Protects kernel from users, but not users

Multilevel security policies

[Feiertag et al., 1977]

- Security level is a pair (A,C) where A is from a totally ordered set (unclassified, …) and C is a set of categories
- Example: data labeled (secret, {nuclear}) is less confidential than (top secret, {nuclear, iraq}) but incomparable to (secret, {iraq})

\[(A_1,C_1) \leq (A_2,C_2) \text{ iff } A_1 \leq A_2 \text{ and } C_1 \subseteq C_2\]

Ordering security policies

[Denning, 1976]

- Information flow policies (security policies in general) are naturally partial orders
  - If policy \( P_2 \) is at least as strong as \( P_1 \), write \( P_1 \) \( \leq \) \( P_2 \)
    - \( P_1 = \) "smoking is forbidden in restaurants"
    - \( P_2 = \) "smoking is forbidden in all public places"
  - Some policies are incomparable:
    - \( P_1 \) \( \not\leq \) \( P_2 \) and \( P_2 \) \( \not\leq \) \( P_1 \)
    - \( P_2 = \) "keep off the grass"

Lattices

- Suppose there is always a least restrictive policy as least as strong as any two policies:
  \( P_1 \sqcup P_2 = \) "join" or least upper bound of \( P_1 \) and \( P_2 \)
  - \( P_1 \sqcup P_2 = \) "smoking is forbidden in restaurants and keep off the grass"
- Simplest policy system is boolean lattice:
  ```
  L \sqcup H, H \sqcup H = H, L \sqcup L = L, L \sqcup H = H
  ```
- If have greatest lower bound too, policies form lattice.
- Supports reasoning about information channels that merge and split
  \((\sqcup=\text{LUB}, \sqcap=\text{GLB})\)

Generalizing levels to lattices

- Security levels may in general form a lattice (or just a partial order)
- \( L_1 \sqsubseteq L_2 \) means information can flow from level \( L_1 \) to level \( L_2 \)
  - \( L_2 \) describes greater confidentiality requirements

\[
\begin{align*}
  c &:= a + b & \quad L_0 \sqcup L_0 \subseteq L_0 \\
  a &:= c & \quad L_0 \subseteq L_0 \sqcap L_0
\end{align*}
\]
Integrity

[Neumann et al., 1976; Biba, 1977]

- Integrity can also be described as a label
- Prevent: bad data from affecting good data
- $L_1 \subseteq L_2$ means information can flow from level $L_1$ to level $L_2$
  - $L_2$ describes lower integrity requirements
  - Lower integrity means use of data is more restricted

- Integrity is dual to confidentiality
  Given: $L_I, H_I$ are low, high integrity
  $L_C, H_C$ are low, high confidentiality
  $L_C \subset H_C$ but $H_I \subset L_I$

Combining properties

- Consider combined policy $(C, I)$ governing both integrity and confidentiality:
  - $(H_C, L)$
  - $(L_C, H_I)$
  - $(L_C, H_C)$
  - $(H_C, H_I)$

Static analysis of information flow

[Denning & Denning, 1977]

- Inference algorithm for determining whether variables are high or low
- Program $\alpha$ enters label tracks implicit flows
  - Computed by dataflow analysis or type system

  $pc = \perp$
  $bool b := <some secret>$
  $if (b) \{ x = true; f(); \}$
  $pc = L_b$

Noninterference

- Low-security behavior of the program is not affected by any high-security data.
  [Cohen, 1977; Goguen & Meseguer 1982]
  - An end-to-end, extensional definition of security

A formalization

- Key idea: behaviors of the system $C$ don’t reveal more information than the low inputs
- Consider applying $C$ to inputs $x$. Define:
  - $[[C]] x$ is the result of $C$ applied to input $x$
  - $s_1 \sim x s_2$ means inputs $s_1$ and $s_2$ are indistinguishable to the low user at level $L$.
  - E.g., $(H, L) \approx_L (H', L')$
  - $[[C]] x_1 \approx_L [[C]] x_2$ means results are indistinguishable:
    low view relation captures observational power

Downgrading & declassification

- Noninterference is too strong
  - Programs release confidential information as part of proper function
- Idea: add escape hatch mechanism that allows system to move data labels downward
- Weakening confidentiality restrictions: declassification
- Example: logging in using a secure password
  - if (password == input) login();
  - else report_failure();
  - Information about the password unavoidably leaks
  - Solution: declassify result of comparison
**Decentralized Label Model**

[ML97]
- Idea: use access control to control what declassifications are allowed
- Principals own parts of labels
- A principal can rewrite its part of the label

\[
\begin{align*}
\text{O}_1: & \text{R}_1, \text{R}_2; \\
\text{O}_2: & \quad \text{R}_2
\end{align*}
\]

- Declassifying code must be trusted by owner
- Other owners' policies still respected

**Intransitive noninterference**

- INI policy augments label lattice with special downgrading arcs
- Password example:
  - Password: label P
  - Other confidential data: label H
  - Public data: label L
- Declassification only allowed along arcs

**Endorsement**

- Dual of declassification: upgrades integrity
- Example: averaging a lot of untrusted data may produce a more trusted result
- Problem: noninterference doesn't hold in presence of downgrading; no equivalently compelling extensional property
  - INI, selective declassification focus attention on security-relevant downgrading operations

**Robust declassification** [ZM01, MSZ04]

- What can we say about end-to-end behavior in presence of declassification?
- One desirable property: untrusted data should not affect what data is released
  - otherwise attackers may be able to control what is released or whether something is released

**Defining robustness**

- Let \( C[a] \) be result of replacing low-integrity code in \( C \) with attack code \( a \). \([C]s\) is result of \( C \) applied to \( s \)
- Robustness:
  \[
  \forall s_1, s_2, a, a'. \quad s_1 =_2 s_2 \Rightarrow \quad [[C[a]]]s_1 =_2 [[C[a']]][s_1]
  \]
  "Attacker learns nothing more by changing attack"
- Can be enforced using static analysis: require inputs to declassification are high integrity
- Qualified robustness permits untrusted sources to affect declassification in limited ways; important for modeling real apps

**Nondeterminism**

- What if the system is nondeterministic?
  - Concurrency: \( (s_1 \land s_2) \rightarrow (s_1' \land s_2) \) or \( (s_1 \land s_2) \)
  - Nondeterministic choice: \( (s_1 \lor s_2) \rightarrow s_1 \) or \( s_2 \)
  - Lack of knowledge about inputs, environment read() \( ? \)
- Noninterference: \( s_1 =_2 s_2 \Rightarrow [[C]s_1] =_2 [[C]s_2] \)
  - What if there are multiple possible results?
Possibilistic security

[Sutherland 1986, McCullough 1987]

• Result of a system \([C]s\) is set of possible outcomes \(\tau\)
  – Outcome could be a trace \(\tau = s \rightarrow s' \rightarrow s'' \rightarrow \ldots\)

• Low view relation on traces is lifted to sets of traces:
  \([C]s_1 \approx_L [C]s_2\) if
  \(\forall \tau_1 \in [C]s_1 . \exists \tau_2 \in [C]s_2 . \tau_1 \approx_L \tau_2\)
  \(\forall \tau_2 \in [C]s_2 . \exists \tau_1 \in [C]s_1 . \tau_2 \approx_L \tau_1\)

"For any result produced by \(C_1\) there is an indistinguishable one produced by \(C_2\) (and vice-versa)"

What is wrong?

• Round-robin scheduler: program equiv. to \(l:=h\)
• Random scheduler: \(h\) most probable value of \(l\)
• System has a refinement with information leak
  \(l := true \mid l := false \mid l := h\)

Example

\(l := true \mid l := false \mid l := h\)

\(h=true: possible results are\)
\(\{h\rightarrow true, l\rightarrow false\}, \{h\rightarrow true, l\rightarrow true\}\)

\(h = false:\)
\(\{h\rightarrow false, l\rightarrow false\}, \{h\rightarrow false, l\rightarrow true\}\)

• Program is possibilistically secure

Conclusions

• Information flow yields a way of talking about end-to-end security properties
• Noninterference: an extensional property enforceable by static analysis
• Neat idea, still not used much in practice
• Some open areas:
  – Dealing with information release
    • Security in the presence of downgrading
    • Connection to access control
  – Information flow in concurrent and distributed systems
  – Application to richer security policies (privacy, anonymity, …)