CS-446/646

Security

C. Papachristos
Robotic Workers (RoboWork) Lab
University of Nevada, Reno

Mandatory Access Control

DAC **vs** ***MAC***

Most people are familiar with *Discretionary Access Control (DAC)*

- Unix *Permission bits* are an example
  - e.g. might set *File* **`private`** such that only *Group* **`friends`** can read it:
  - **`-rw-r----- 1 dm friends 1254 Feb 11 20:22 private`**
- Anyone with access to information can further propagate that information at their discretion:
  - **`$ mail sigint@enemy.gov < private`**

Mandatory Access Control (MAC) can restrict Propagation

- *Note:* MAC here not be confused with *Message Authentication Codes* or *Medium Access Control*
- Security administrator may allow you to read but not disclose *File*

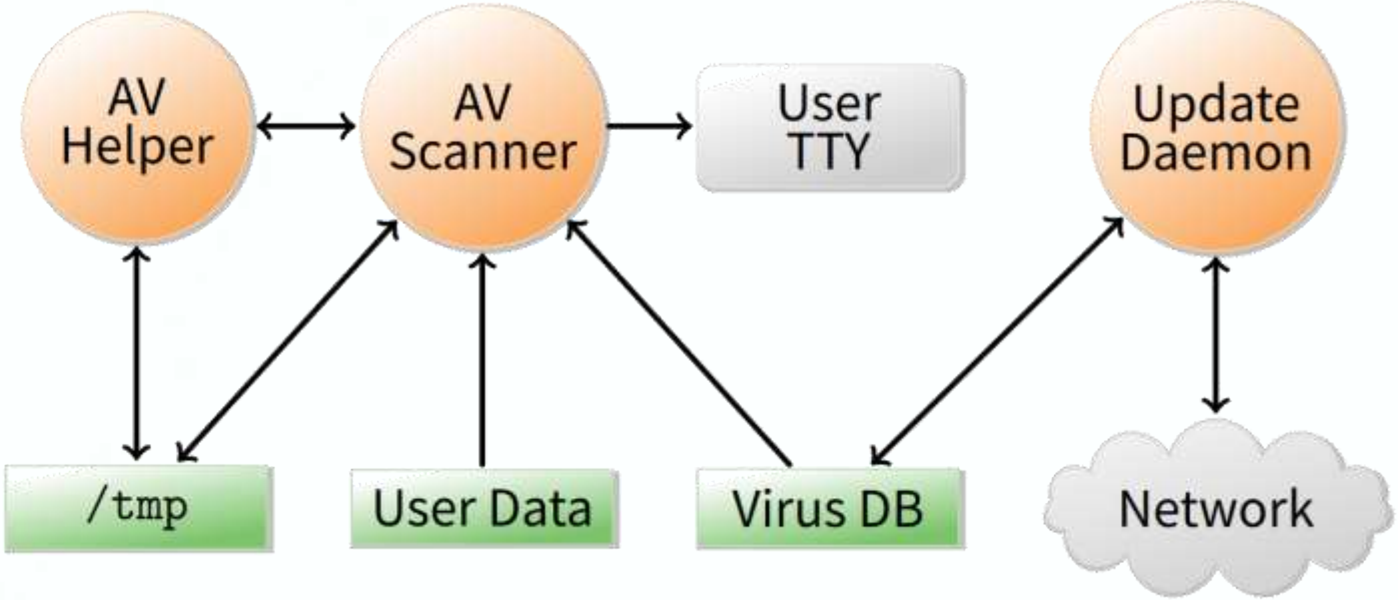
Mandatory Access Control

***MAC* Motivation**

- Prevent users from disclosing sensitive information (whether accidentally or maliciously)
  - e.g. Classified information requires such protection
- Prevent Software from surreptitiously leaking data
  - Seemingly innocuous Software may steal secrets in the background
  - Such a program is known as a "*Trojan Horse*"
- Case study: *Symantec AntiVirus 10*
  - Contained a "*Remote Exploit*" (attacker could run arbitrary code)
  - Inherently required access to all of a user's *Files* to scan them
  - Can an OS protect private (e.g. Classified) *File* contents under such circumstances?

Mandatory Access Control

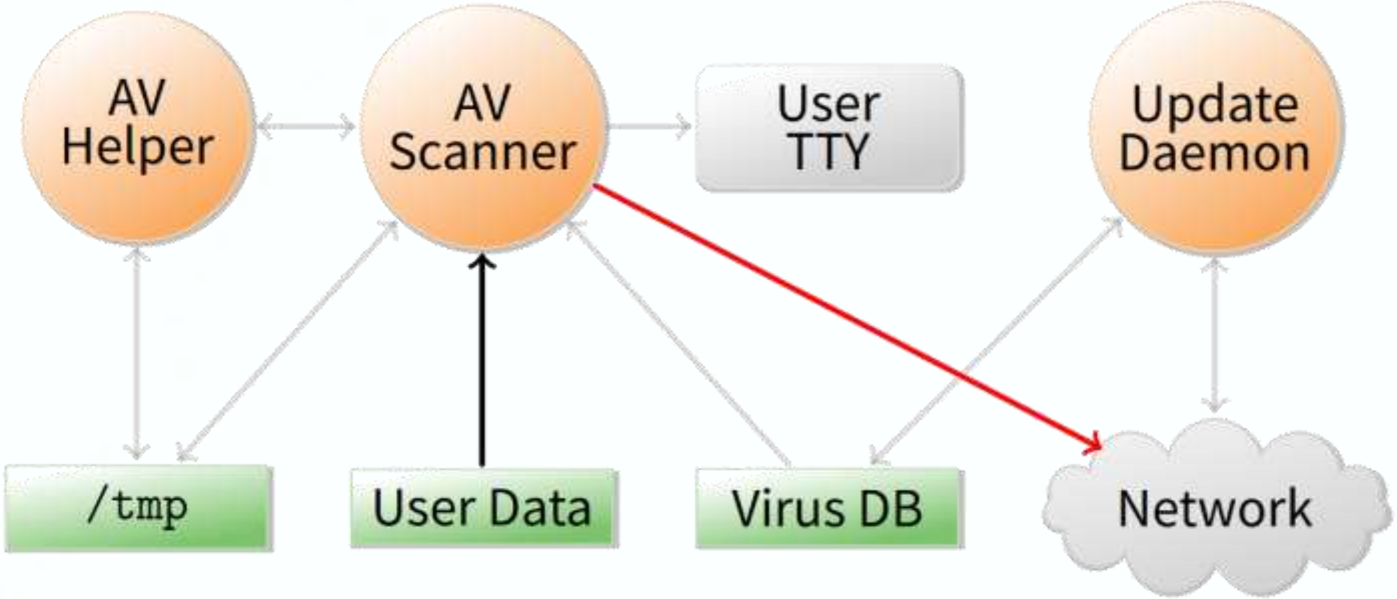
Example: ***Anti-Virus (AV)*** **Software**



- *Scanner* : Checks for Virus Signatures
- *Update Daemon* : Downloads new Virus Signatures
- How can OS enforce *Security* without blindly trusting AV Software?
  - Must not leak contents of your *Files* to Network
  - Must not tamper with contents of your *Files*

Mandatory Access Control

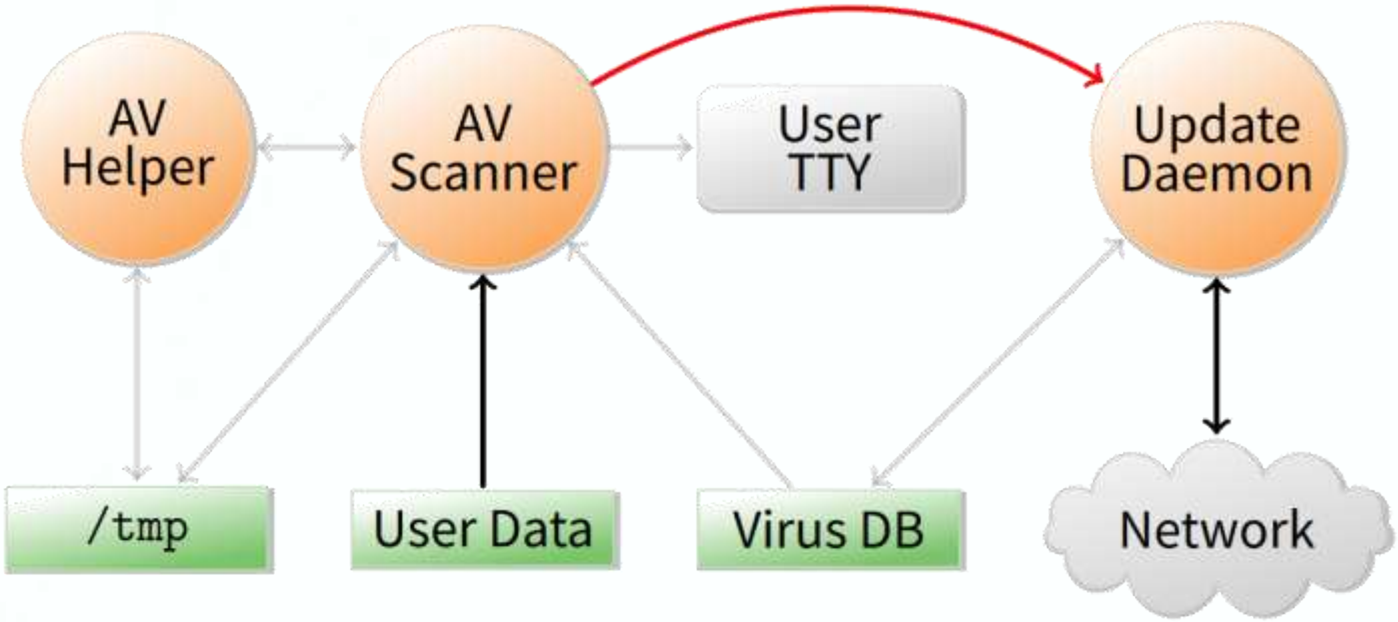
Example: ***Anti-Virus (AV)*** **Software**



- *Scanner* can write your private data to Network
- Prevent *Scanner* from invoking any *System Call* that might send a Network message?

Mandatory Access Control

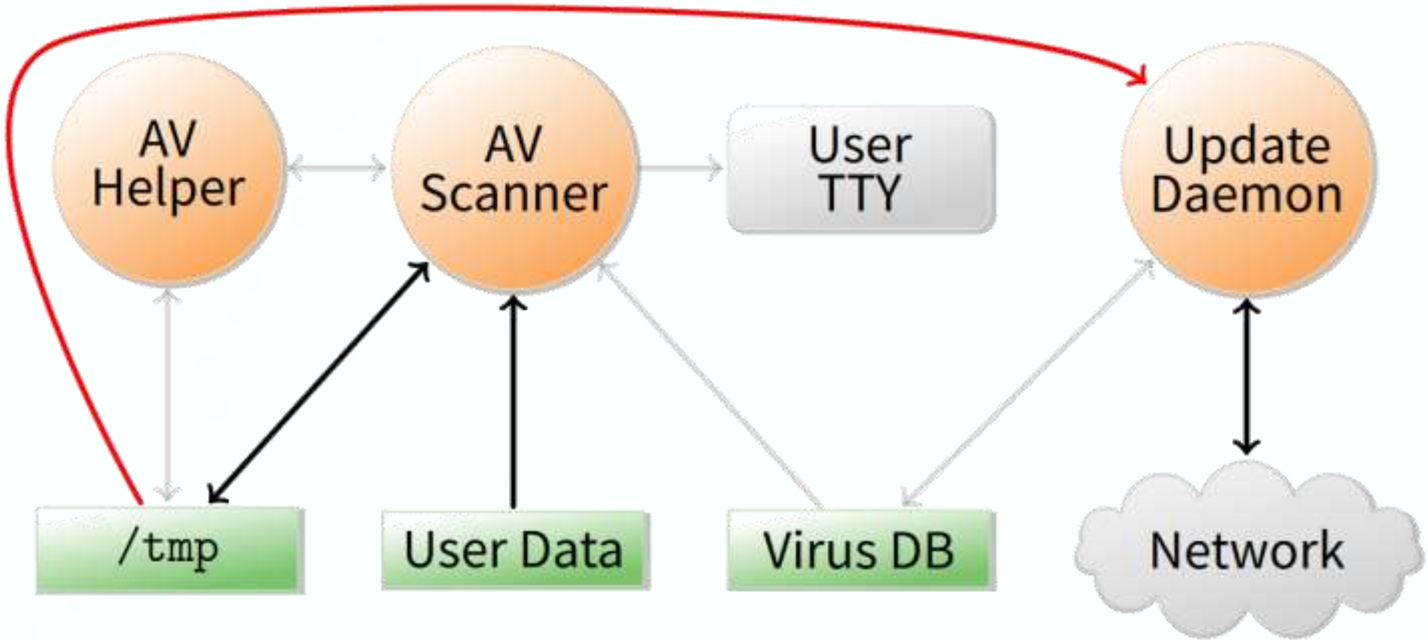
Example: ***Anti-Virus (AV)*** **Software**



- *Scanner* can send private data to *Update Daemon*
- *Update Daemon* sends data over Network
  - Can cleverly disguise secrets in order/timing of update requests
- Block *Inter-Process Communication* & *Shared Memory*–related *System Calls* in *Scanner*?

Mandatory Access Control

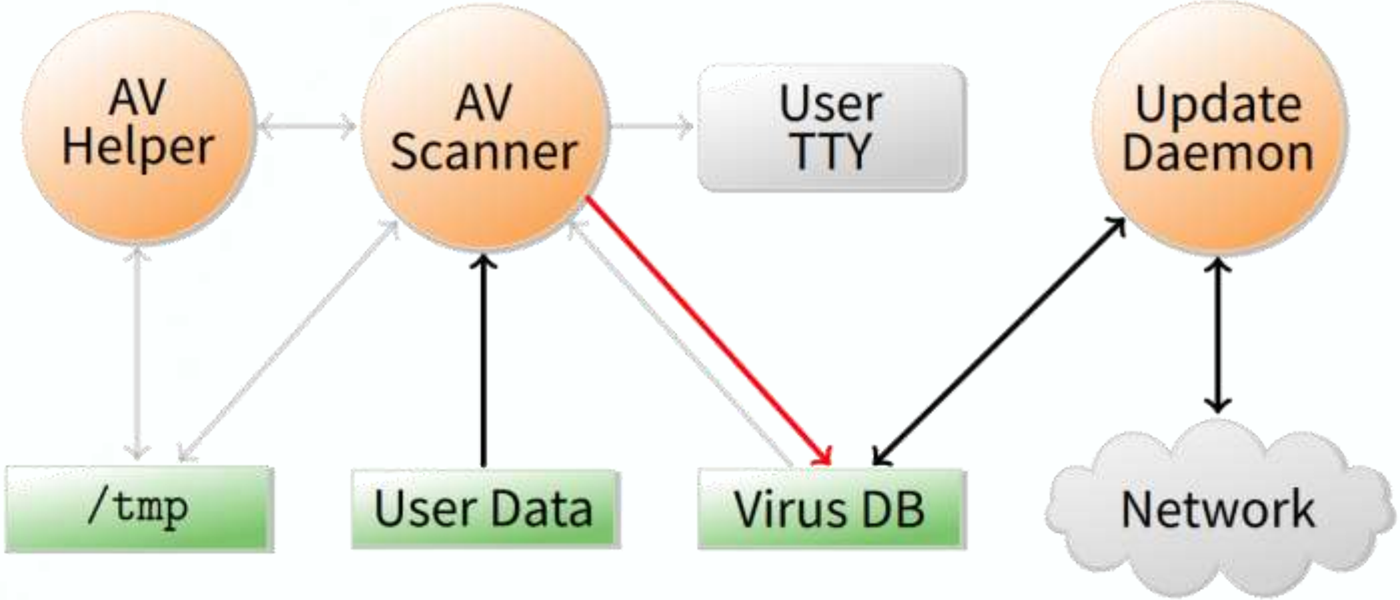
Example: ***Anti-Virus (AV)*** **Software**



- *Scanner* can write data to world-readable *File* in **`/tmp`**
- *Update Daemon* later reads and discloses *File*
- Prevent *Update Daemon* from using **`/tmp`**?

Mandatory Access Control

Example: ***Anti-Virus (AV)*** **Software**



- *Scanner* can acquire read *Locks* on Virus Signature Database
  - **Encode** private User data by *Locking* various ranges of *File*
- *Update Daemon* **decodes** data by detecting *Locks*
  - Then disclose private User data over the Network
- Have *Trusted* Software copy Virus DB for *Scanner*?

Mandatory Access Control

The list goes on…

- *Scanner* can call **`setproctitle`** with User data
  - *Update Daemon* can then extract User data by running **`ps`**
- *Scanner* can **`bind`** particular TCP or UDP *Port* numbers
  - Sends no Network traffic, but used *Port* numbers detectable by *Update Daemon*
- *Scanner* can relay data through another *Process*
  - Call **`ptrace`** to take over *Process*, then write to Network
  - Use **`sendmail`**, **`httpd`**, or **`portmap`** to reveal data
- Disclose data by modulating free *Disk* space
- Can be certain we've covered all possible communication channels?
  - Not without a more systematic approach to the problem

Labels and Lattices

Secrecy / Confidentiality

- Achieve Controlled Access to Classified information

Typical way to think of *Security* in this context:

- A subject at a given *Security Level* should not be able to read information at higher *Security levels*, **and**
- A subject at a given *Security Level* should not be able to write information (leak it) at lower *Security levels*

- I.e. a User can create content only at or above their own *Security Level* (e.g. a Secret-level researcher can create Secret-level or Top-Secret-level *Files* but may not create Public-level *Files*). Conversely, a User can view content only at or below their own *Security Level* (e.g. a Secret-level researcher can view Public-level or Secret-level *Files*, but may not view Top-Secret-level *Files*).

- Transfer of information from a high-Secrecy document to a lower-Secrecy document may happen via *Trusted* subjects only

Bell-La Padula model [BL]

- View the system as Subjects accessing Objects
  - *Access Control* : Take Requests as input and output Decisions
- Four modes of Access are possible:
  - $execute$: No Observation or Alteration
  - $read$: Observation
  - $append$: Alteration
  - $write$: Both Observation and Alteration
- An *Access Matrix* M encodes permissible *Access* modes
  - Subjects are rows, Objects are columns
- The current *Access Set* : b is (*Subj*, *Obj*, *Attr*) : (S, O, A) triples
  - Encodes Accesses in progress (e.g. open *Files*)
  - At a minimum, $(S, O, A) \in b$ requires A permitted by cell $M_{S,O}$

Labels and Lattices

Security Levels

- A *Security Level* (or *Label*) L_i is a pair $\langle c, s \rangle$ where:
  - c = *Classification* – e.g. 1 = Unclassified, 2 = Secret, 3 = Top-Secret
  - s = *Category-Set* – e.g. Nuclear, Crypto
- $\langle c_1, s_1 \rangle$ "*Dominates*" $\langle c_2, s_2 \rangle$ *iff* $c_1 \geq c_2$ *and* $s_1 \supseteq s_2$
  - L_1 *Dominates* L_2 relation is sometimes written $L_1 \propto L_2$ or $L_1 \sqsupseteq L_2$
  - *Labels* then form a *Lattice* (partial order with "least upper-bound" & "greatest lower-bound")
- Inverse of "*Dominates*" relation is : "*Can Flow To*", written $\sqsubseteq$
  - $L_1 \sqsubseteq L_2$: L_1 *Can Flow To* L_2 (means that L_2 *Dominates* L_1)
- Subjects and Objects are assigned *Security Levels*
  - $Level(S), Level(O)$: *Security Level* of Subject/Object
  - $CurrentLevel(S)$: Subject may operate at a lower *Security Level*
  - $Level(S)$ *Bounds* $CurrentLevel(S)$: $CurrentLevel(S) \sqsubseteq Level(s)$
  - Since $Level(S)$ is max, sometimes called S's "*Clearance*"

Security Properties

- Two Access Control *Properties* with respect to *Security Labels*:

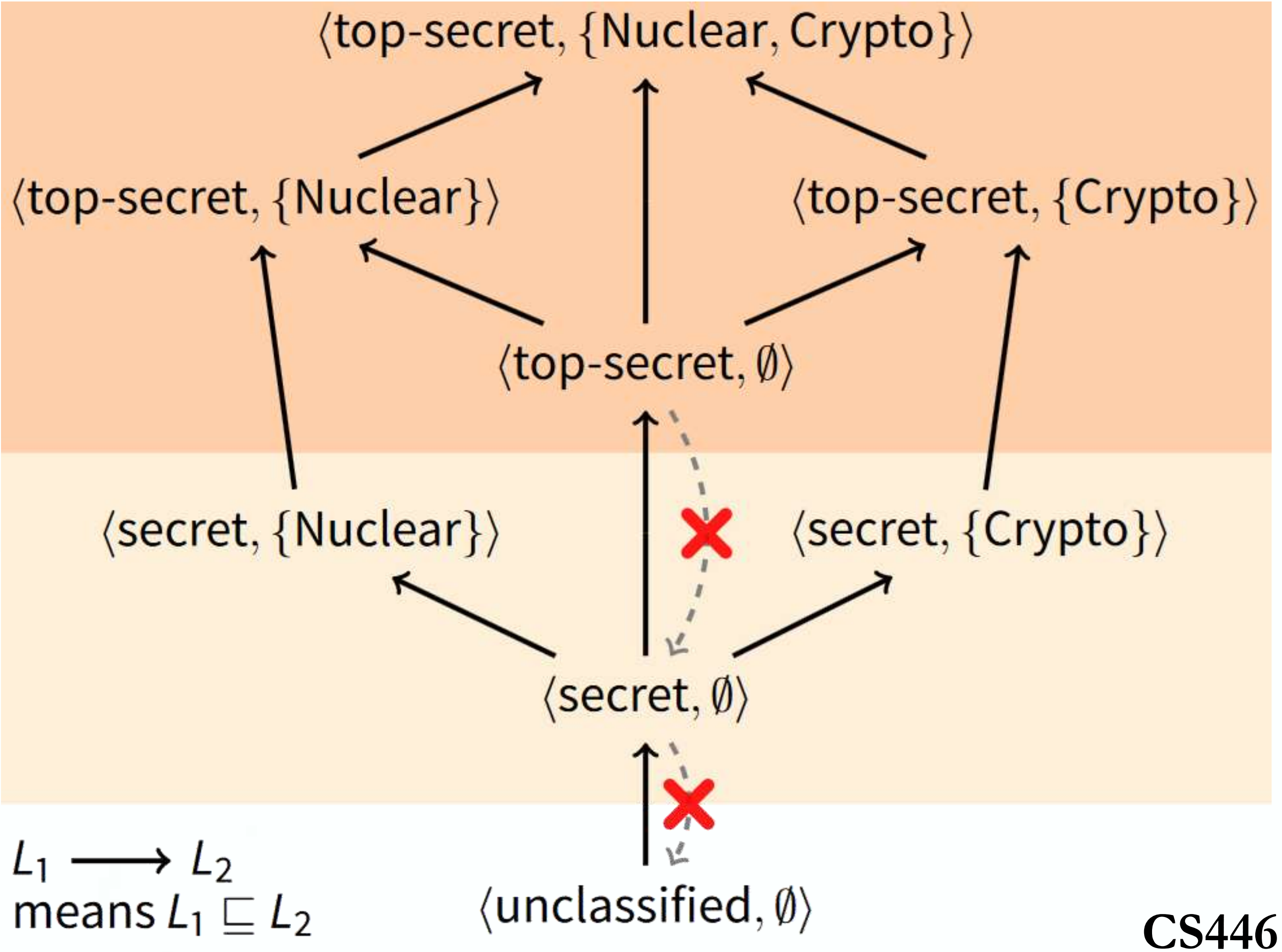
1. *Simple Security* or *SS-Property* (DAC)
   - For any $(S, O, A) \in b$, if A includes $observation$, then $Level(O) \sqsubseteq Level(S)$ (i.e. $Level(S)$ must *Dominate* $Level(O)$)
     - e.g. an Unclassified User cannot $read$ a Top-Secret *File* – "*No Read-Up*"

2. *Star Security* or $*$-*Property* (MAC)
   - If any $Subject$ $observes$ O_1 and $modifies$ O_2, then $Level(O_1) \sqsubseteq Level(O_2)$ (i.e. $Level(O_2)$ *Dominates* $Level(O_1)$)
     - e.g. no $Subject$ can $read$ a Top-Secret *File*, then $write$ a Secret *File* – "*No Write-Down*"

- More precisely, given an $(S, O, A) \in b$:
  - if $A = r$ then $Level(O) \sqsubseteq CurrentLevel(S)$ – "*No Read-Up*"
  - if $A = a$ then $CurrentLevel(S) \sqsubseteq Level(O)$ – "*No Write-Down*"
  - if $A = w$ then $CurrentLevel(S) = Level(O)$

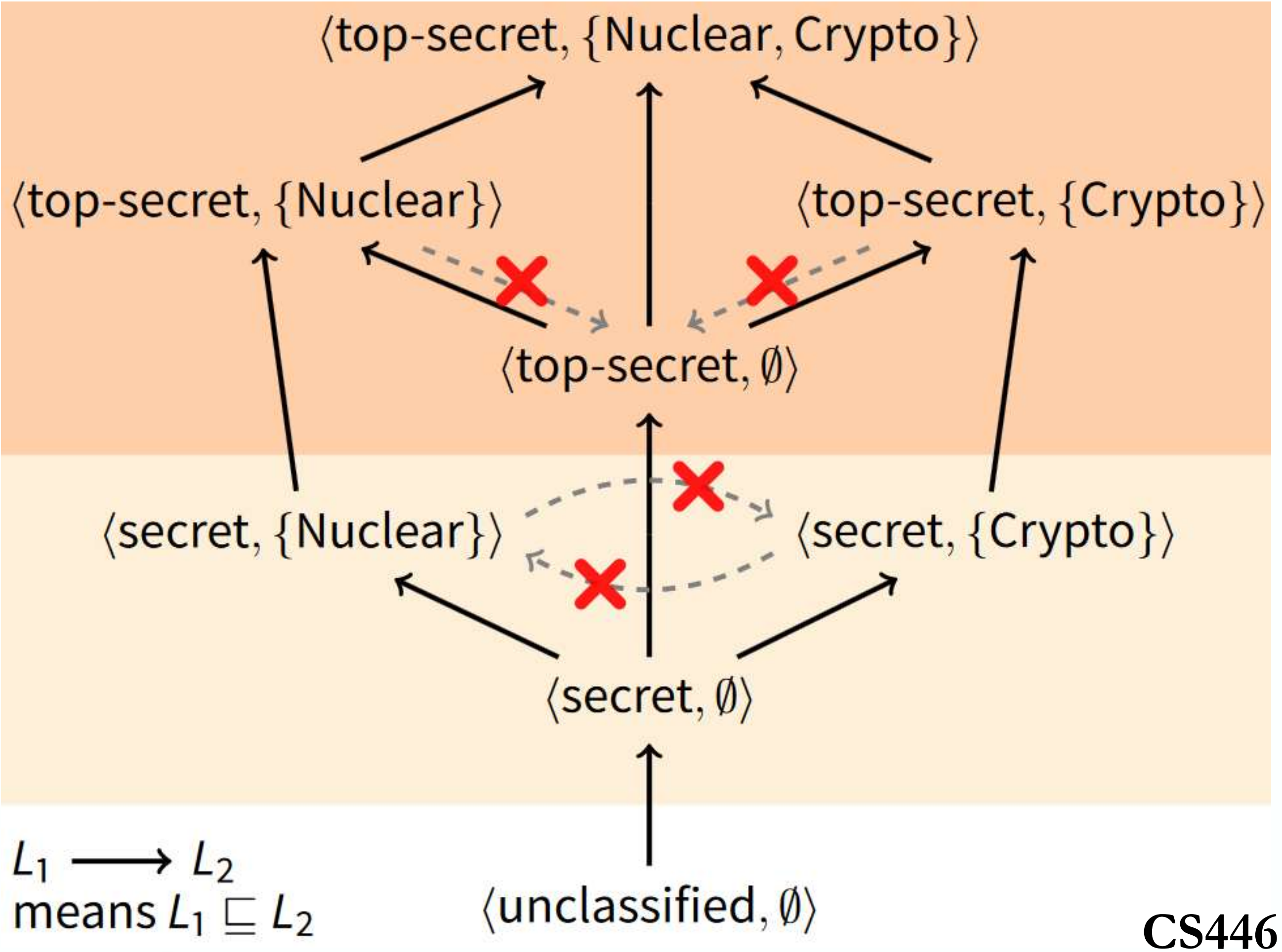
Labels and Lattices

Labels **form a *Lattice* [Denning]**



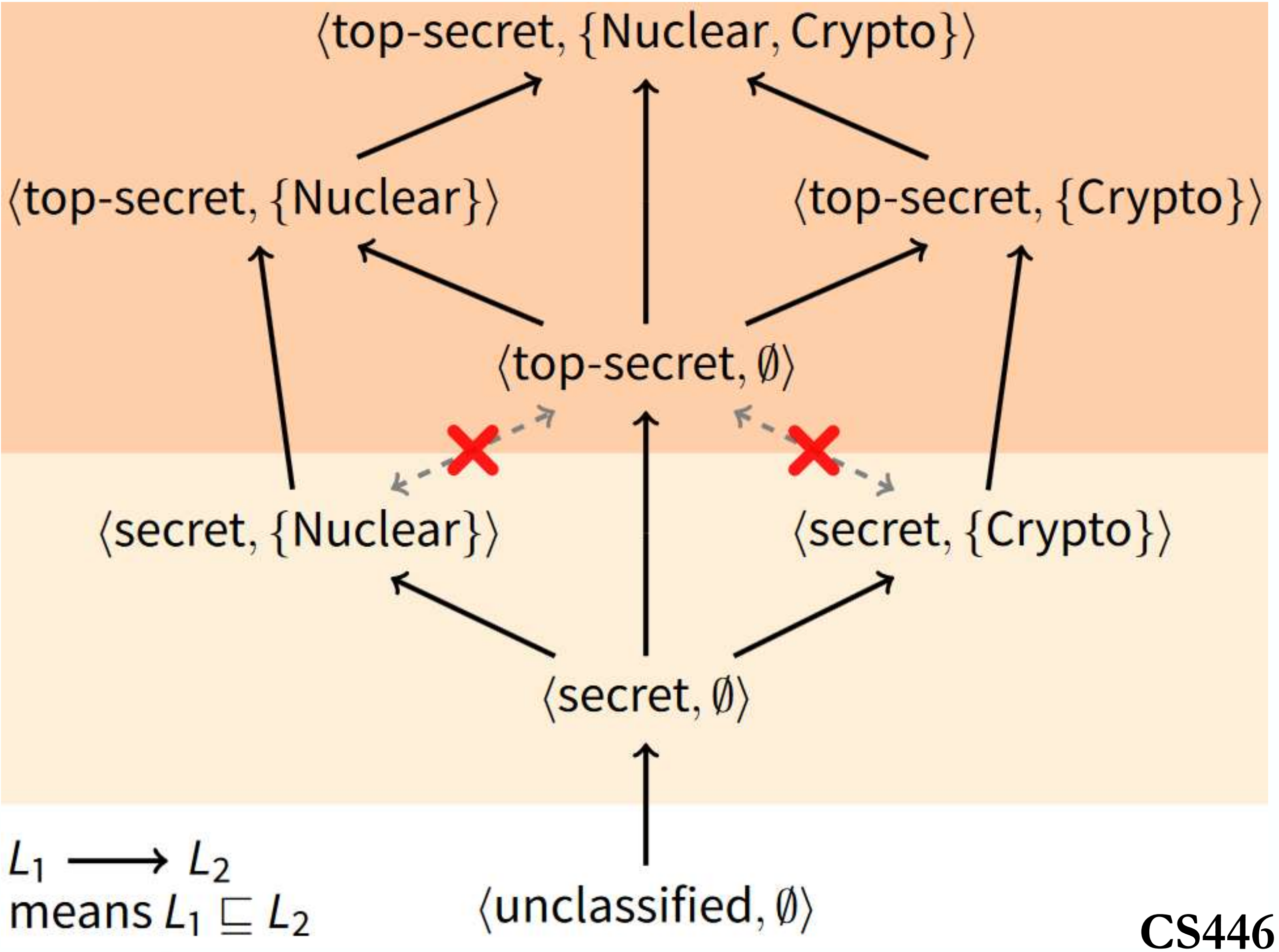
Labels and Lattices

Labels **form a** ***Lattice*** **[[Denning]]**

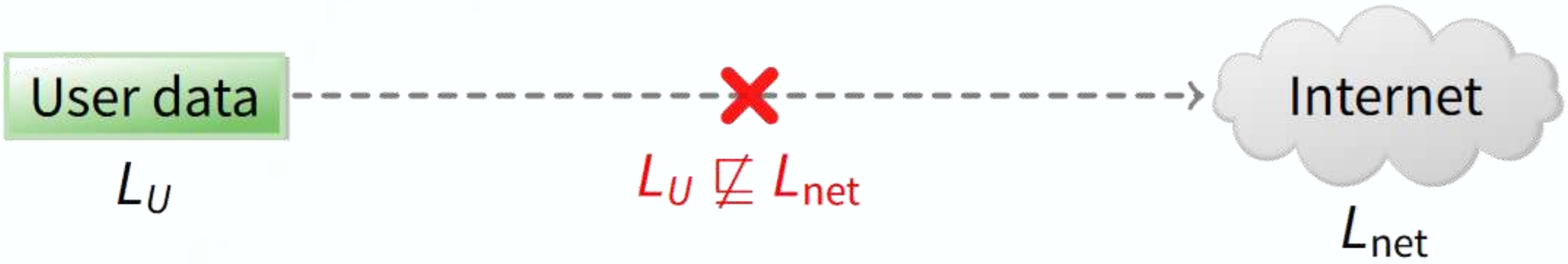


Labels and Lattices

Labels **form a** ***Lattice*** **[[Denning]]**



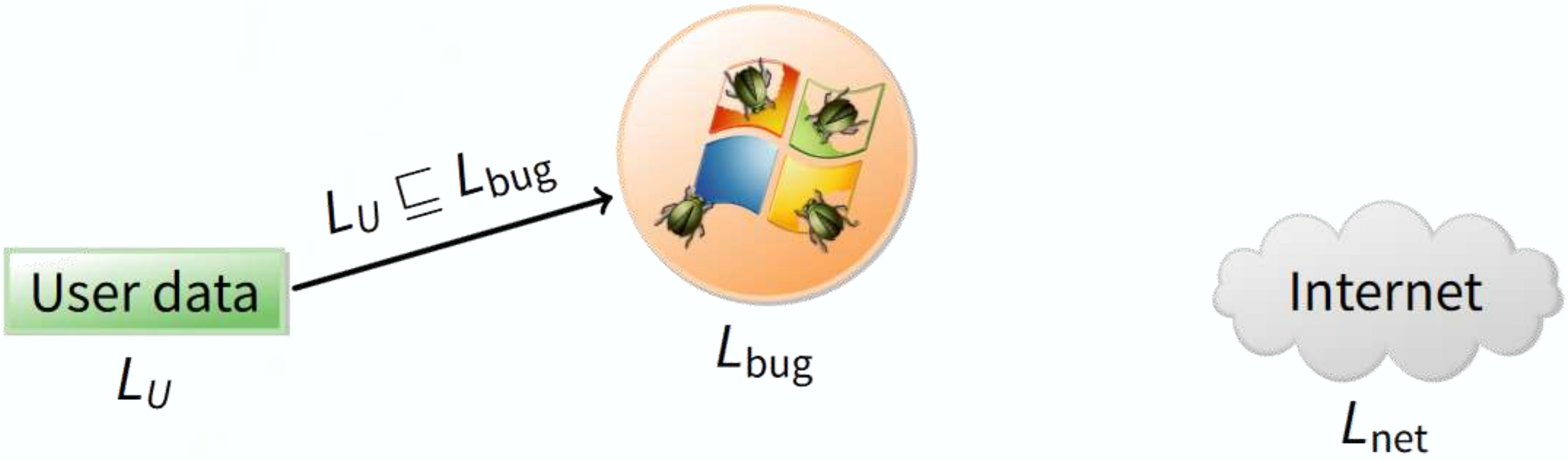
$\sqsubseteq$ is ***Transitive***



- *Transitivity* makes it easier to reason about *Security*
- Example: *Label* User data (*File*) so it cannot *Flow-To* Internet: $L_U \not\sqsubseteq L_{net}$
  - Policy holds regardless of what other Software does

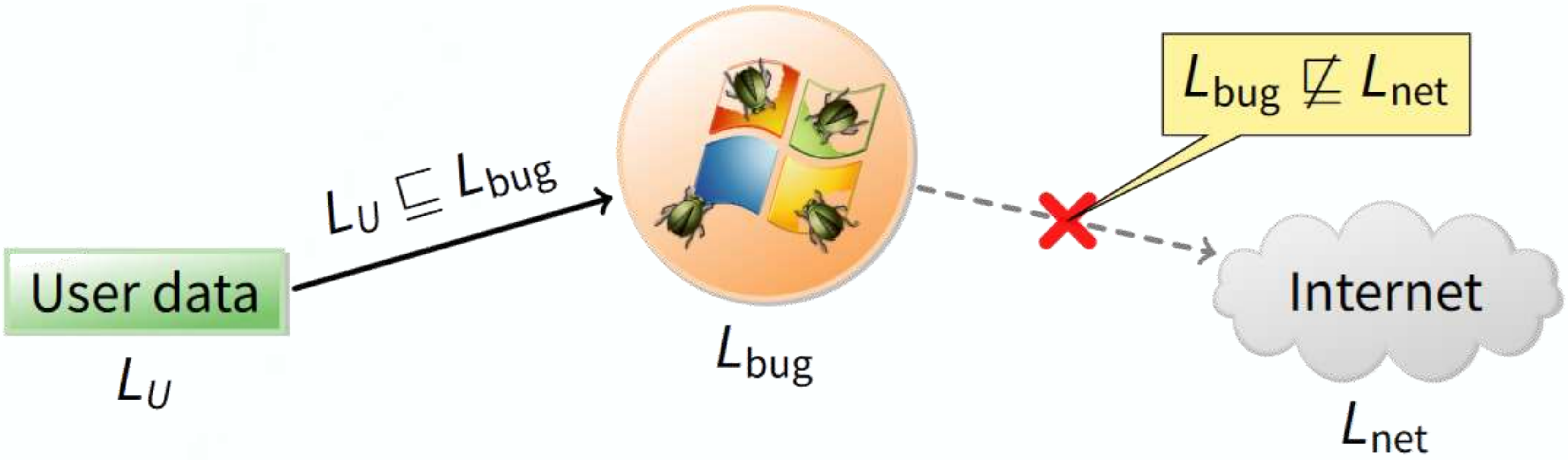
Labels and Lattices

$\sqsubseteq$ is ***Transitive***



- *Transitivity* makes it easier to reason about *Security*
- Example: *Label* User data (*File*) so it cannot *Flow-To* Internet: $L_U \not\sqsubseteq L_{net}$
  - Policy holds regardless of what other Software does
- Suppose untrustworthy Software ***reads*** *File*

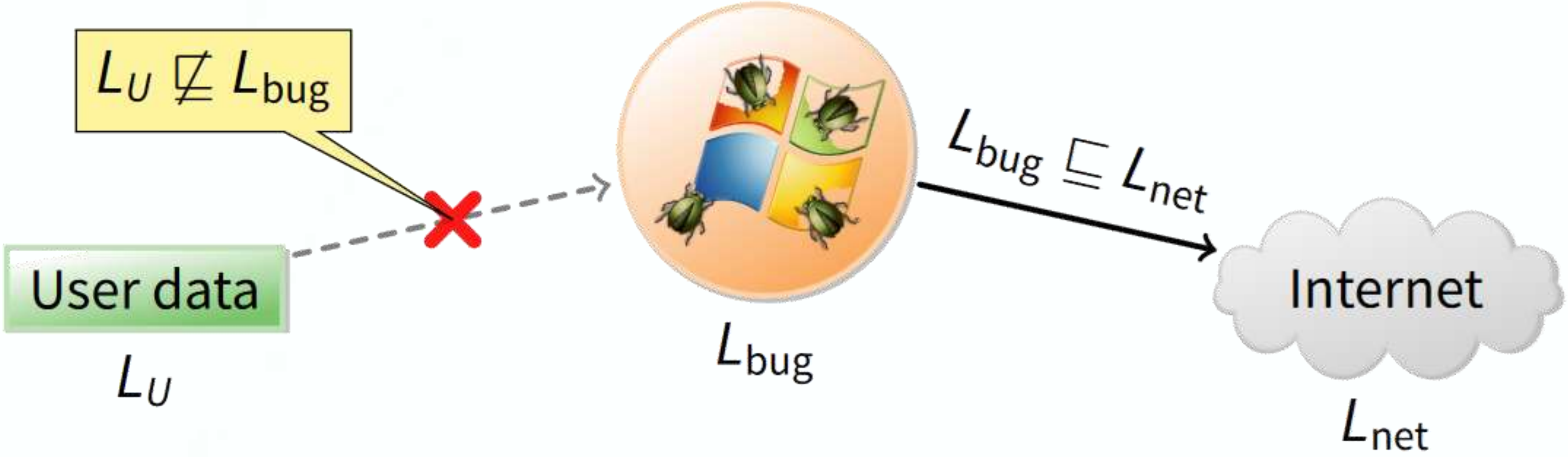
$\sqsubseteq$ is ***Transitive***



- *Transitivity* makes it easier to reason about *Security*
- Example: *Label* User data (*File*) so it cannot *Flow-To* Internet: $L_U \not\sqsubseteq L_{net}$
  - Policy holds regardless of what other Software does
- Suppose untrustworthy Software ***reads*** *File*
  - *Process* that is *Labeled* L_{bug} ***reads*** *File*, so must have $L_U \sqsubseteq L_{bug}$
  - If $L_U \sqsubseteq L_{bug}$ and $L_U \not\sqsubseteq L_{net}$, it follows that $L_{bug} \not\sqsubseteq L_{net}$

Labels and Lattices

$\sqsubseteq$ is ***Transitive***



- *Transitivity* makes it easier to reason about *Security*
- Example: *Label* User data (*File*) so it cannot *Flow-To* Internet: $L_U \not\sqsubseteq L_{net}$
  - Policy holds regardless of what other Software does
- Conversely, a *Process* that can $write$ to the Network cannot $read$ the *File*

Naïve MAC Implementation

- Take an ordinary Unix system
- Put *Labels* on all *Files* and *Directories* to track *Security Levels*
- Each User U assigned a *Security Clearance* $\mathtt{Level}(U)$ on login
- Determine $\mathtt{Current}$*(Security)*$\mathtt{Level}$ dynamically
  - When U logs in, start with lowest $\mathtt{CurentLevel}$
  - Increase $\mathtt{CurentLevel}$ as higher-level *Files* are $\mathtt{observed}$
    - (sometimes called a "*Floating Label*" system)
  - If U's *Level* does not *Dominate* $\mathtt{CurentLevel}$, kill program
    - e.g. kill Program that $\mathtt{writes}$ to *File* if the Program's $\mathtt{CurentLevel}$ can't *Flow-To* the *File*'s *Label*

- Is this *Secure*?

No: *Covert Channels*

- Operating System rife with *Covert Storage Channels*
  - *Low* `CurrentLevel` Process executes another Program
  - New Program reads sensitive *File*, gets *High* `CurrentLevel`
  - *High* Program can then exploit *Covert Channels* to pass data to *Low* `CurrentLevel` *Process*
  - Example: *High* Program inherits *File Descriptor*
    - Can pass 4-Byte information chunks to *Low* Program through the ***File* offset** !
- Other *Covert Storage Channels*:
  - **exit value, signals, *File* locks, terminal escape codes,** …
- If we eliminate all *Covert Storage Channels*, is system *Secure*?

Still No: *Timing Channels*

- *Example*: **CPU Utilization**
  - To send a 0 bit, use 100% of CPU in busy-loop
  - To send a 1 bit, sleep and relinquish CPU
  - Repeat to transfer more bits
- *Example*: **Resource Exhaustion**
  - *High* Program allocates all Physical Memory if bit is 1
  - If *Low* Program slows down due to *Paging*, knows less Memory available
- Other *Examples*:
  - **Disk head position**, **Processor Cache/TLB pollution**, …

Reducing *Covert Channels*

- Observation: *Covert Channels* come from *Sharing*
  - If you have no *Shared Resources*, no *Covert Channels* exist
  - Extreme example: Just use two computers (common in DoD)
- Problem: *Sharing* is needed
  - e.g. ***read*** Unclassified data when preparing (***writing***) Classified
- In general, can only hope to bound Bandwidth of *Covert Channels*
- One approach: **Strict *Partitioning* of *Resources***
  - Strictly *Partition* and *Schedule Resources* between *Levels*
  - Occasionally reapportion *Resources* based on usage [Browne]
  - Do so infrequently to bound leaked information
  - Approach still not so good if many *Security Levels* possible

Labels and Lattices

Declassification

- Sometimes need to prepare Unclassified report from Classified data
- *Declassification* happens outside of traditional Access Control Model
  - Present *File* to security officer for downgrade
- Job of *Declassification* often not trivial
  - e.g., Microsoft Word saves a lot of Undo information
    - This might be all the secret stuff you cut from document
  - Another bad mistake:
    - Redact PDF using black censor bars over or under text, leaving text selectable
      - e.g. [Cluley]

Labels and Lattices

Integrity

- Achieve Controlled Access to Classified information

Typical way to think of *Security* in this context:

- A Subject should not be able to corrupt Data in an *Integrity Level* ranked higher than themself
- A Subject should not be able to become corrupted by Data from a lower *Integrity Level*

Preservation of data *Integrity* – Goals:

- Prevent Data modification by unauthorized parties
- Prevent unauthorized Data modification by authorized parties
- Maintain Internal and External Consistency (i.e. Data reflects the real world)

Biba Integrity Model [Biba]

- Problem: How to protect *Integrity*
  - Suppose text editor gets Trojan-ed, subtly modifies *Files*
  - Might mess up critical operations even without leaking anything
- Observation: *Integrity* is the converse of *Secrecy*
  - In *Secrecy*, want to avoid writing to *Lower-Secrecy Files* – Prevent *Leaking*
  - In *Integrity*, want to avoid writing *Higher-Integrity Files* – Prevent *Corruption*

Use *Integrity* hierarchy parallel to *Secrecy* one

- Now *Label* (/*Security Level*) is a $\langle c, \boldsymbol{i}, s \rangle$ triple, where $\boldsymbol{i}$ = *Integrity*
- $\langle c_1, \boldsymbol{i_1}, s_1 \rangle \sqsubseteq \langle c_2, \boldsymbol{i_2}, s_2 \rangle$ *iff* $c_1 \leq c_2$ *and* $\boldsymbol{i_1} \geq \boldsymbol{i_2}$ *and* $s_1 \subseteq s_2$
  - Only *Trusted* Users can operate at *Higher Integrity* (which is visually lower in the *Lattice* – Opposite of *Secrecy*)
  - If you $\boldsymbol{read}$ less authentic data, your $\boldsymbol{CurrentIntegrityLevel}$ gets lowered (placing you higher in the *Lattice*), and you can no longer $\boldsymbol{write}$ *Higher-Integrity Files*

LOMAC

LOMAC **[Fraser]**

- MAC not widely accepted outside military

Low water Mark Access Control (LOMAC)'s goal:

  - Make MAC more palatable

- Concentrates on *Integrity*
  - More important goal for many settings
    - e.g. don't want Viruses tampering with all your *Files*
  - Also don't have to worry as much about *Covert Channels*
- Provides reasonable defaults (minimally obtrusive)
- Has actually had impact
  - Originally available for Linux (2.2)
  - Now ships with FreeBSD
  - Windows introduced similar *Mandatory Integrity Control (MIC)*

LOMAC Overview

Subjects are *"Jobs"* (essentially *Processes* – possibly groups of *Processes;* more later)

- Each Subject labeled with an *Integrity* number (e.g. 1, 2)
- Higher numbers mean higher *Integrity* (so –unfortunately– 2 $\sqsubseteq$ 1 using earlier notation)
- Subjects can be *Reclassified* (change $CurrentIntegrityLevel$) on $observation$ of *Low-Integrity* data

Objects (*Files*, *Pipes*, etc.) also labeled with *Integrity Level*

- Object *Integrity Level* is fixed and cannot change

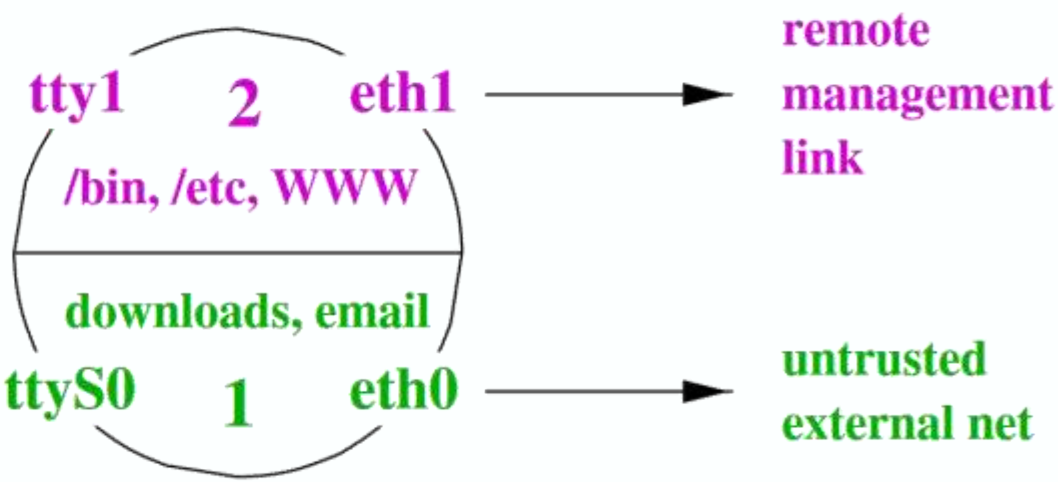
In the context of Security:

- *Low-Integrity* Subjects cannot write to *High-Integrity* Objects
- New Objects acquire *Integrity Level* of their creator

LOMAC

LOMAC Defaults

Note: *Can-Flow-To* is downward;
opposite of earlier diagram



- Two *Integrity Levels*: 1 and 2
- Level 2 (*High-Integrity*) contains:
  - FreeBSD/Linux *Files* intact from distro, static Web Server config
  - The console, *Trusted* terminals, *Trusted* Network
- Level 1 (*Low-Integrity*) contains:
  - NICs connected to Internet, *Untrusted* terminals, etc.
- Idea: Suppose Worm compromises your Web Server
  - Worm comes from external Network → Level 1
  - Won't be able to muck with System *Files* or Web Server config

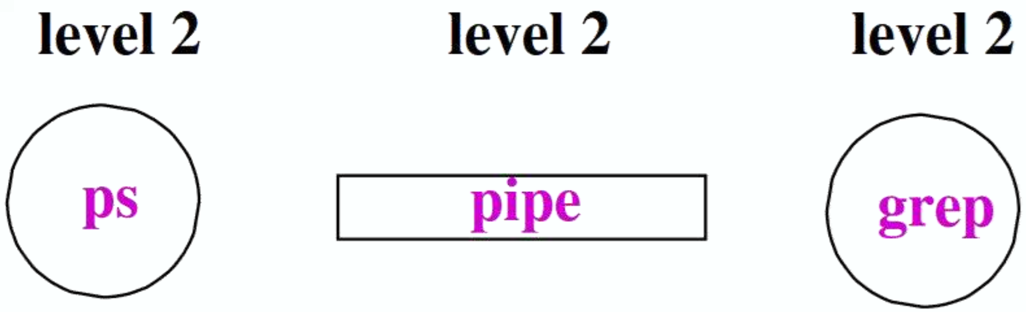
LOMAC

The *Self-Revocation* Problem

- Want to integrate with Unix unobtrusively
- Problem: Application expectations
  - Kernel performs Access Checks usually at *File* ***open*** time (i.e. not ***read*** / ***write***)
    - *Remember: Access Control Lists & Capabilities (File Descriptors)*
  - Legacy Applications don't pre-declare they will ***observe*** *Low-Integrity* data
  - An Application can "*taint*" itself unexpectedly, *Revoking* its own Permission to access an Object it created

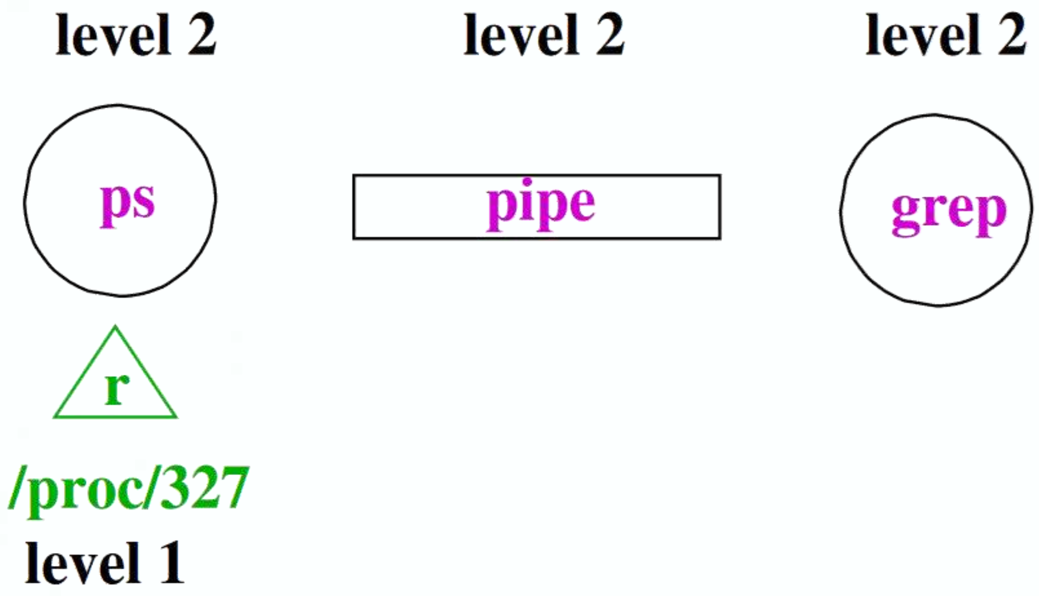
Self-Revocation Example

- User has *High-Integrity* (Level 2) Shell
- Runs: **`ps | grep user`**
  - *Pipe* created before **`ps`** *`reads`* *Low-Integrity* data
  - **`ps`** becomes "*tainted*", can no longer *`write`* to **`grep`**



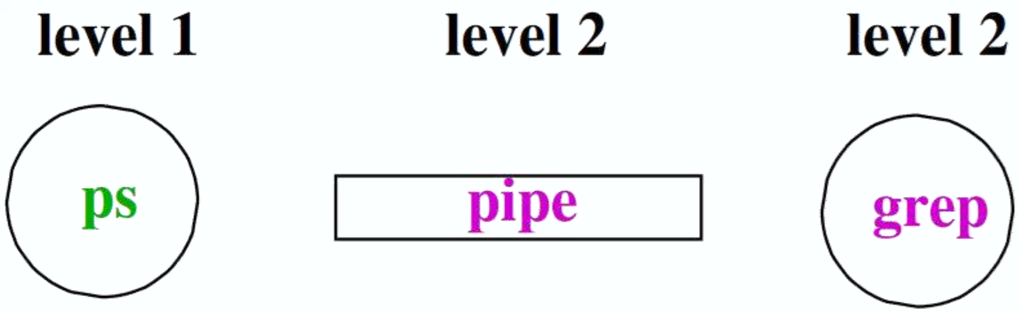
Self-Revocation Example

- User has *High-Integrity* (Level 2) Shell
- Runs: **`ps | grep user`**
  - *Pipe* created before **`ps`** *`reads`* *Low-Integrity* data
  - **`ps`** becomes "*tainted*", can no longer *`write`* to **`grep`**



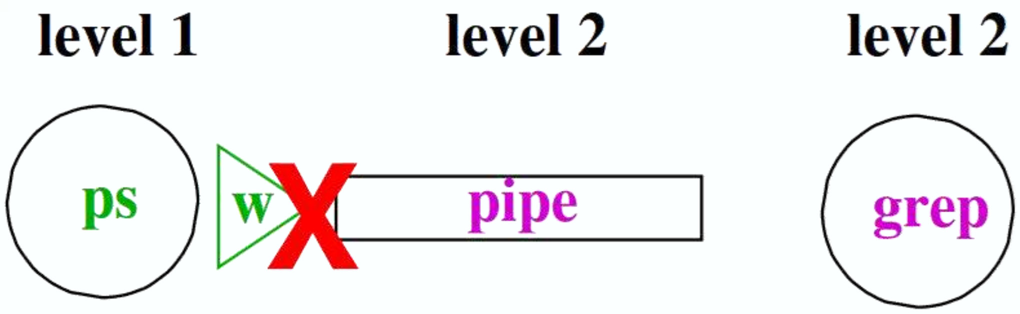
Self-Revocation Example

- User has *High-Integrity* (Level 2) Shell
- Runs: **`ps | grep user`**
  - *Pipe* created before **`ps`** *`reads`* *Low-Integrity* data
  - **`ps`** becomes "*tainted*", can no longer *`write`* to **`grep`**



Self-Revocation Example

- User has *High-Integrity* (Level 2) Shell
- Runs: `ps | grep user`
  - *Pipe* created before **ps** *reads* *Low-Integrity* data
  - **ps** becomes "*tainted*", can no longer *write* to **grep**



LOMAC

Solution

- Don't consider Pipes to be real Objects
- Join multiple *Processes* together in a "*Job*"
  - Pipe ties *Processes* together in *Job*
  - Any *Processes* tied to *Job* when they ***read*** or ***write*** to Pipe
  - So will lower *Integrity* of both **`ps`** and **`grep`**
- Similar idea applies to *Shared Memory* and *Inter-Process Communication*
- Summary: LOMAC applies MAC to non-military systems
  - But doesn't allow military-style Security policies
    - i.e. with *Secrecy*, various *Categories*, etc.

CS-446/646

Time for Questions !