Bell-La Padula model

- **Simple security condition:** $S$ can read $O$ if and only if $l_O \leq l_S$

- ***-property:** $S$ can write $O'$ if and only if $l_S \leq l_O$

- “Read down; write up”
  - Information flows upward

- Why?
  - Information flow
  - Could be due to a malicious insider, or a benign mistake
Basic security theorem

- If the Bell–La Padula rules are enforced, then no information in an object at level \( l_O \) can leak into an object at level \( l_O' < l_O \)

Communicating down...

- How to communicate from a higher security level to a lower one?
  - (Not necessarily declassification; instead, moving unclassified data from a classified machine to an unclassified machine)

- Max. security level vs. current security level
  - Maximum security level must always dominate the current security level
  - Reduce security level to write down...
    - Security theorem no longer holds
    - Must rely on users to be security-conscious
Commercial vs. military systems

- The Bell-LaPadula model does not work well for commercial systems
  - Users should be given access to data as needed
    - Discretionary access control vs. mandatory access control
  - Would require using a large number of categories and classifications
  - Requires centralized handling of “security clearances”
  - Poor usability

Biba model

- Concerned with integrity
  - “Dual” of Bell-LaPadula model
- The higher the level, the more confidence
  - More confidence that a program will act correctly
  - More confidence that a subject will act appropriately
  - More confidence that data is trustworthy
- Integrity levels may be independent of security levels
  - Confidentiality vs. trustworthiness
  - Information flow vs. information modification
Biba model

- $I_S, I_O$ denote integrity levels

- **Simple integrity condition**: $S$ can write $O$ if and only if $I_O \leq I_S$
  - The information obtained from a subject cannot be more trustworthy than the subject itself

- **(Integrity) *-property**: $S$ can read $O$ iff $I_S \leq I_O$
  - $S$ should depend on higher-quality sources only

- “Read up; write down”
  - Information flows downward

Security theorem

- An information transfer path is a sequence of objects $o_1, \ldots, o_n$ and subjects $s_1, \ldots, s_{n-1}$, such that, for all $i$, $s_i$ can read $o_i$ and write to $o_{i+1}$
  - Information can be transferred from $o_1$ to $o_n$ via a sequence of read-write operations

- Theorem: If there is an information transfer path from $o_1$ to $o_n$, then $I(o_n) \leq I(o_1)$
  - Informally: information transfer does not increase the trustworthiness of the data

- Note: says nothing about secrecy…
Chinese wall

- Intended to prevent conflicts of interest
  - E.g., consulting firms
- Rights are dynamically updated based on actions of the subjects

Chinese wall

- Objects are grouped into datasets (e.g., all files associated with some client)
- Datasets are groups into conflict-of-interest (CoI) classes (e.g., all datasets related to banks)
Chinese wall -- basic setup

Company datasets

Bank A  Bank B  School 1  School 2

School 3

Conflict of interest (CoI) class

Chinese wall rules

- Subject S is allowed to read from at most one company dataset in any CoI class
  - This rule is dynamically updated as accesses occur
  - See next slide...

- Formally: S can read from dataset X iff it has not previously read from any other dataset in the same CoI class as X
**Example**

- Bank A
- Bank B
- School 1
- School 2
- School 3

**Chinese wall rules II**

- S can write to dataset X only if
  - S can only read from dataset X

- Note: either S cannot write at all, or can only write to one dataset

- This is intended to prevent an indirect flow of information that would cause a conflict of interest
  - E.g., S reads from Bank A and writes to School 1; S’ can read from School 1 and Bank B
  - S’ may find out information about Banks A and B!
Role-based access control

- Controls based on a user’s (current) role, not their identity

- Users assigned to different roles
  - Can encode this in a matrix with users as rows, and roles as columns
  - A user can have multiple roles assigned
  - User can select current role based on their current function

- Access controls based on roles
  - Can use a “access matrix”, where subjects are roles
RBAC: basic idea

Users

Roles

Resources

- research
- marketing
- admin

Server 1
Server 2
Server 3

Advantages

- Users change more frequently than roles, RBAC has lower administrative overhead
- More compact than a full-blown access control matrix
- Least privilege – users can take on roles as needed
Groups vs. roles

- A group is a (largely static) set of principals
- A role defines a set of access permissions
  - Different users take on different roles at different times
- Military analogy:
  - The group of users who are sergeants
  - The role of “person on watch duty”

Buffer overflows
Buffer overflows

- Previous focus in this class has been on secure policies and mechanisms
- For real-world security, security of the policy / mechanism is not enough -- the implementation must also be secure
  - We have seen this already when we talked about side-channel attacks
  - Here, the attacks are active rather than passive
  - Also, here the attacks exploit the way programs are run by the machine/OS

Importance of the problem

- Most common cause of ‘technical’ attacks
  - Over 50% of CERT advisories related to buffer overflow vulnerabilities
- Morris worm (1988)
  - 6,000 machines infected
- CodeRed (2001)
  - 300,000 machines infected in 14 hours
- Etc.
Buffer overflows

- Fixed-sized buffer that is to be filled with unknown data, usually provided directly by user
- If more data “stuffed” into the buffer than it can hold, that data spills over into adjacent memory
- If this data is executable code, the victim’s machine may be tricked into running it
- Can overflow on the stack or the heap…

A glimpse inside a computer

![Diagram of computer memory and registers]

- Registers
  - EBP
  - ESP
  - EIP

- Function frame
- Stack
- Heap/memory
- Code
Registers

- Special memory adjacent to the CPU
  - Provides extremely fast access
  - But only a limited number of registers

- Some are general purpose, some are specific

- Three specific ones we will care about:
  - EIP (instruction pointer, aka program counter)
  - ESP (stack pointer)
  - EBP (frame pointer)

Code

- Instructions to be run, written in machine language
  - Machine specific
  - We assume x86 architecture here

- The EIP register contains the address of the next instruction to be executed
  - Typically, as each instruction is executed the EIP is incremented by the appropriate amount
  - EIP can also change based on control flow (e.g., a JMP or a function call)
Stack

- Each function that is executed is allocated its own frame on the stack
  - This frame stores local variables, function parameters
  - Also stores EIP/EBP

- When one function calls another, a new frame is initialized and placed (pushed) on the stack

- When a function is finished executing, its frame is taken (popped) off the stack

- Traditionally, the stack grows toward lower memory addresses

Frames and function calls
Stacks/frames

- The ESP always contains the address of the top-most entry on the stack

- When a value is *pushed* to the stack
  - The value is placed directly above the top-most entry, and the ESP is decremented

- When a value is *popped* from the stack
  - The value is read, and the ESP is incremented
  - The value is not deleted from memory (but it will be overwritten as later values are pushed to the stack)
    - This can be a potential security vulnerability as well

---

Stacks/frames

- The EBP contains the address of the bottom of the current frame
  - I.e., the address of the saved EBP

- The EBP only changes when a frame is pushed on / popped off the stack, not when individual values are pushed/popped
Function calls I

- EIP pointing to instruction “CALL <address>”
  - The EIP of the instruction to execute after the call returns (the saved EIP) is pushed on the stack
  - EIP changed to the address of the called code

- Function preamble
  - The current EBP (the saved EBP) is pushed to the stack
  - The EBP is set equal to the current ESP
  - The ESP is decremented to leave enough space on the stack for the current function frame

Function calls II

- Function leave/return
  - The ESP is set equal to the EBP
    - The has the effect of popping the current frame
  - Pop the stack and set the EBP to the popped value
    - This was the saved EBP
  - Pop the stack and set the EIP to the popped value
    - This was the saved EIP
Buffer overflows

- The problem is that some functions that read user input are “unsafe”
  - E.g. (in C), gets, strcpy read input until the first null character – even if the destination buffer is too small!
  - Results from one buffer “overflow” into the other

“Simple” buffer overflow

- Overflow one variable into another

- gets(color)
  - What if I type “blue 1”?
  - (Actually, need to be more clever than this)
More devious examples...

- `strcpy(buf, str)`

- What if `str` has more than `buf` can hold?
  - Problem: `strcpy` does not check that `str` is shorter than `buf`

- Overwrite EIP, change expected control flow of program (cf. in-class examples)

Examples using gdb