ENTS 689i
Part II: System Security

Lecture 6: OS Security
Today's Class

• Two components to OS security
  – Protecting user programs
  – Protecting the OS itself
• We're going to focus mostly on the first
  – Authentication
  – Access control
• Will also touch on some of the second
  – OS design
  – OS bugs
What is an operating system?

• In the old days, there was no OS
  – Users entered programs into a queue
  – Batch processing loaded and ran the program
  – User provided entire environment

• Then there were small OSes
  – Provided easier transition between users
  – And some common tools, like libraries
  – But no “online” contention for resources

• Today: systems are multiuser/program
  – Separated and managed by OSes
Typical OS responsibilities

• Resource allocation
  – Memory
  – Disk/storage
  – Processor time (scheduling)

• Input/output

• Protection
  – Keep user's resources safe from other users

• Sharing
  – Safe access to shared resources
Process abstraction

- **Process**: instance of a running program
  - Same program can have multiple processes
- Run on behalf of users
- Make kernel requests for resources
  - I/O
  - More memory
  - Creating other processes
- Protected from other processes
  - Can't modify each other directly
- Kernel protected from processes
Memory Protection
Separation

• Basic form of protection
• Keep user's objects completely isolated
• Several possible types:
  – Physical – different memory, devices, etc.
  – Temporal – same resources, different times
  – Logical – appears there are no other resources
  – Cryptographic – can't interpret data of others
• Secure, but impractical
  – What about sharing?
Fences

- Simple boundary between user/kernel
- Used in single-user OSes

Address limit register

Operating System

Access Denied

User Program
Fences

• Can use a corresponding bounds register
  – Dynamic offset and length
• With single pair, can't protect user code
  – Whole program is in same protection domain
  – Program may accidentally overwrite itself
• Fixable with second register pair
  – Code in one fence, data in another
• What if we want to have more policies?
  – e.g., code, read-only data, writable data
  – Program needs to be modified
Segmentation

- Generalization of fences
- Divide program into arbitrary-sized sections
- Each has a unique name
- Program addressed using <name,offset>
- OS has table of names, mappings, accesses
- HW assists in enforcing access protection
- $\geq 1$ process can be allowed access to a segment
  - With same or different privileges
- OS can swap segment, load on use
- Must check segment bounds on each access
Segmentation

Logical Program Organization

- Code
- Data Seg 1
- Data Seg 2
- Data Seg 3

Physical Memory

- Data Seg 2
- Other programs
- Data Seg 1
- Other programs
- Code
- Data Seg 3
Paging

- Divide program into fixed-size **pages**
- Divide memory into **page frames**
- Addresses are <page,offset>
- OS maintains map of pages to page frames
  - Called **page tables**
  - Typically hierarchical
- More efficient division of memory
  - No fragments, unlike segmentation
- Transparent to user
  - No logical structure, unlike segmentation
Paging

Logical Program Organization

- Page 0
- Page 1
- Page 2
- Page 3
- Page 4
- Page 5

Physical Memory

- Page 0
- Page 3
- Page 2
- Page 1
- Page 4
- Page 5
Memory protections in practice

- Most modern processors support paging
- Some support segmentation
- x86 supports both
- Most x86 OSes use “flat” segmentation
- Each system can have different controls
  - Until recently, no NX bit on x86 paging
- Invalid accesses result in “faults”
  - OS handles condition based on error code
  - Fault could mean “swap” or “bad access”
Linux process protections

- Each process has own set of page tables
- Believes it has full $2^{32}$ addresses
  - Actually mapped into different physical pages
- Other procs not mapped into same PTs
  - Therefore, no access to those procs
  - Exception: a few shared data pages
- OS mapped into PTs of all processes
  - Same virtual addresses for each
Linux kernel protections

• Processors support multiple privilege levels
  – x86 has 4 rings (0-3)
  – Only 2 typically used (0,3)
• OS can use privilege levels for isolation
• User code cannot read/write kernel pages
• Kernel code can read/write all pages
  – Kernel write can be disabled with optional flag
• Processor provides several mechanisms to “call” into the kernel at well-defined entries
Linux x86 process layout
Access Control
Access control

• Memory is not the only thing to protect
• More generally, want to protect objects
  – Inactive pieces of data
• Accessed by subjects: active entities
  – Users and their processes
• Support different accesses for each user
• But who should have which accesses?
  – Matter of policy
• And how can it be enforced?
  – Matter of mechanism
Access control

• Typical examples of controlled objects:
  – Files
  – I/O devices
  – Other processes

• Typical accesses:
  – Read
  – Write
  – Execute
  – Ownership (also called “grant/revoke”)
### Access control matrix

<table>
<thead>
<tr>
<th></th>
<th>File A</th>
<th>Temp File</th>
<th>Myfile</th>
<th>compiler</th>
<th>linker</th>
<th>CLOCK</th>
<th>PRINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>USER A</td>
<td>ORW</td>
<td>ORW</td>
<td>ORW</td>
<td>X</td>
<td>X</td>
<td>R</td>
<td>W</td>
</tr>
<tr>
<td>USER B</td>
<td>R</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>R</td>
<td>W</td>
</tr>
<tr>
<td>USER S</td>
<td>RW</td>
<td>-</td>
<td>R</td>
<td>X</td>
<td>X</td>
<td>R</td>
<td>W</td>
</tr>
<tr>
<td>USER T</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>R</td>
<td>W</td>
</tr>
<tr>
<td>SYSTEM</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>OX</td>
<td>OX</td>
<td>ORW</td>
<td>O</td>
</tr>
<tr>
<td>SERVICES</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>R</td>
<td>W</td>
</tr>
</tbody>
</table>

**Read:** R  
**Write:** W  
**Execute:** X  
**Own:** O
Access directories

• Per-user list of objects and permissions
  – One list for each user (single row of matrix)
  – Maintained by the OS
• Easy to implement
• But there are some challenges
  – For shared accesses, too many duplicates
  – Even if user doesn't intend to access them
  – Revoking all accesses very expensive
  – Duplicate file names confusing to track
Access control lists

• Rather than a list per-user, track per-file
  – Single column of matrix
• List of <user, permission> pairs for each file
• Enables default entries for all users
  – More efficient for public files
• Revocation is more efficient
  – Simply need to traverse one list
Capabilities

• Alternative to complete OS access tracking
• Provide each subject with a capability
  – Unforgeable token that certifies access
• Use protected OS calls to generate tokens
• Subjects can grant tokens to other subjects
• In some implementations, subject can define new accesses
Capabilities

• Protection can take many forms
  – Direct HW support (requires extra memory)
  – Give subject an index into protected OS table
  – Encrypt token before giving to subject

• Revocation is harder
  – Kernel must track outstanding capabilities

• Example: Unix file descriptors
  – open() returns an integer index
  – corresponds to a file object in kernel
  – reads/writes pass index back to kernel
Discretionary vs. Mandatory

• So far, we have seen examples of **discretionary** access control (DAC)
  – Owner decides who has which accesses
  – OS enforces owner's wishes
• Stronger systems employ **mandatory** access control (MAC)
  – Central authority determines valid accesses
• Some systems employ both
  – MAC policy checked first
  – If passes, check for more restrictive DAC policy
Role-based access control

- Rather than focus on users, focus on tasks
- Define a set of roles with access rights
- When users perform task for that role, act with those rights
- Some users will have different roles at different times
- Can revoke rights when role is complete
- Often associated with user groups
  - e.g., administrators, power users, security officers
Military security policy

• Goal: protect classified information

• Hierarchical **levels** of sensitivity:
  – unclassified < confidential < secret < top secret

• Some data also protected by nonhierarchical **compartments**
  – e.g., {project A}, {project B}

• Data **classification**: <level, compartments>

• Subjects are granted **clearances** also of <level, compartments>
Military security policy

• Relation $\leq$, **dominance**

\[
o \leq s \text{ if and only if } \\
\text{rank}_o \leq \text{rank}_s \text{ and } \\
\text{compartments}_o \subseteq \text{compartments}_s
\]

• “$s$ has access to $o$ if his clearance is at least as high as the classification of $o$ and he has all of the necessary compartments”

• If so, we say “$s$ dominates $o$”
Bell LaPadula

- Formal **model** of military security policy
- Intended to ensure confidentiality
  - Information never flows where it should not
- Set of subjects $S$, objects $O$
- Each has a fixed security class $C(s)$ or $C(o)$
- Ordered with dominance relation
Bell LaPadula

• **Simple security property:**
  – A subject $s$ may have read access to an object $o$ only if $C(o) \leq C(s)$

• ***-Property:**
  – A subject $s$, with read access to an object $o$ may have write access to an object $p$ only if $C(o) \leq C(p)$

• “Read down, write up”
Biba's “strict integrity”

• Dual of Bell LaPadula, for integrity
• Integrity levels $I(o)$ and $I(s)$, analogous to sensitivity levels for confidentiality
  – Based on trustworthiness of entity
• Simple integrity policy:
  – A subject $s$ can modify object $o$ only if $I(s) \geq I(o)$
• Integrity *-property:
  – If $s$ has read access to object $o$ then $s$ can have write access to $p$ only if $I(o) \geq I(p)$
Clark-Wilson

- For commercial purposes, integrity is at least as important as secrecy
- Want to make sure data is only updated according to well-formed transactions
- Identify key constrained data items (CDIs)
  - And set of constraints for data consistency
- Test constraints using integrity verification procedures (IVPs)
- Define trusted transformation procedures
  - Ensure data is consistent after transformation
TPs are written carefully and verified
  – Only certifier decides who can run TP
Only TPs transform CDIs
  – Valid state to valid state
Example: Bank transactions
  – $D = \text{today's deposit}$
  – $W = \text{today's withdrawals}$
  – $TB = \text{today's balance}$
  – $YB = \text{yesterday's balance}$
Constraint: $TB = YB + D - W$
Chinese Wall

- Certain commercial situations need to avoid conflicts of interest
- For example, companies with customers who are competitors
  - Users with access to one shouldn't have access to others
- Divide data into conflict classes
  - Users have access to $\leq 1$ item in each class
- Labels are dynamic
  - Given one access, other accesses are revoked
Real Access Control Systems

- Unix filesystem
- SELinux
- AppArmor
- LoMAC
  - Implementation of Biba's low water mark
- BSD Jails/secure levels
Unix filesystem controls

• Files organized into hierarchical directories
  – /home/npetroni/mydir/myfile
• Files have an associated set of permissions for three entities: owner, group, others
• Permissions: read (4), write (2), execute (1)
• Straightforward for files
• Directories a little more complicated
  – Read: list file names in directory
  – Execute: traverse to that directory
  – Write: modify entries in directory
Unix filesystem controls

• Each user and group have an identifier
• Permission are not inherited from path
• Instead, each user has a **mask** of default creation permissions

• **Setuid** bit:
  – Run executable with uid of file owner

• **Setgid** bit:
  – Run executable with gid of file group
  – For directories, new entries inherit gid instead of creator's default gid
Authentication
Authentication Principles

• Something you **KNOW**
  – Typical example: a password

• Something you **POSSESS**
  – Called a **token**

• Something you **ARE**
  – Or how you behave

• Combinations of the above
Factors in authentication design

• Resistance to counterfeiting/circumvention
• Time to authenticate/burden on users
• Cost of installation/upkeep
  – Direct equipment costs
  – Administrative costs (e.g., adding a new user)
• Reliability and Accuracy
  – False acceptances: invalid users get access
  – False rejections: valid users don't
Password authentication

• Most common form today

• Conventional wisdom, two key factors:
  – Hard to guess
  – Easy to remember

• Password vulnerabilities:
  – Common words or names
  – Easy to obtain information
    • Where have we seen this recently?
  – Keyboard patterns
  – Password reuse

Simple permutations of the above:
Password checking

• Password must stored be on the system
  – Otherwise, how could it be checked?

• Can be stored in different forms:
  – Cleartext
  – Dedicated server
  – Encrypted
  – Hashing

• Passwords are often stored with other info
  – Which users need to be able to change/view
  – But that reduces security on the PW

Solution: shadow (separate) password file
One-time passwords

• Challenge-response
  – System presents a challenge
  – Serves as input to a secret function
  – Answer computed by user and returned

• SecureID
  – Password rotates every 30 seconds
  – Based on function with shared secret
  – User carries token with synchronized clock
  – Sometimes protected with PIN
Biometrics

• Authenticate user based on physical characteristics of user
• Examples
  – Retina
  – Facial recognition
  – Voice
  – **Fingerprints**
  – Typing/writing characteristics
Biometrics

- Require physical device, may be costly
- Variability in sampling introduces errors
  - System has to approximate
  - False positives: model is too loose
  - False negatives: model doesn't account for natural variability (or the thing really changed)
- Sometimes slower to authenticate
  - Trade-off with number of samples
- Doesn't prevent all forgeries
- Have human/social issues as well
Design principles
OS Design

• OSes are obviously important
• In real life OSes are very complex
  – Tens of millions of lines of code
• Bugs in OSes are a big deal
  – Compromise may result in loss of protections
• High-level goal:
  – Want some assurance that the design and implementation are correct
• Design is (or should be) influenced by likelihood of verification being feasible
Trusted computing base

• Everything necessary to enforce the security policy of the system

• Non-TCB components cannot negatively impact system security
  – Even if attacker controls them

• Goal: small subset of OS and hardware
  – But in practice, this is a large part of the system
  – Includes trusted userland programs, e.g., login
Reference monitor

• A small part of the system is responsible for enforcing all accesses
• Typically part of the OS kernel
• Can be spread across different components
• Must have the following properties:
  – **Tamperproof**: impossible to disable
  – **Unbypassable**: all other access impossible
  – **Analyzable**: small enough to be tested and scrutinized