What is a trusted system?

• “Secure” tends to be an absolute
• “Trusted” means that it meets the necessary security requirements
  ◆ May not be completely secure!
  ◆ Needs to justify the user’s confidence!
• Trust comes in degrees
  ◆ Trust best friends with deep, dark secrets
  ◆ Trust acquaintances with my plans for this weekend
  ◆ Trust strangers to give me directions (usually…)

CMPS 122, UC Santa Cruz

Trusted Operating Systems
Trusted systems vs. secure systems

<table>
<thead>
<tr>
<th>Secure system</th>
<th>Trusted system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary: it’s secure, or it’s not</td>
<td>Degrees of trustworthiness</td>
</tr>
<tr>
<td>Intrinsic to the system</td>
<td>User decides on trustworthiness</td>
</tr>
<tr>
<td>Based solely on the system</td>
<td>User makes a judgment based on system</td>
</tr>
<tr>
<td>Absolute: security doesn’t depend on how or by whom the system is used</td>
<td>Relative: trustworthiness depends on the details of system use</td>
</tr>
<tr>
<td>Goal: absolute security</td>
<td>Characteristic: system can be viewed as trustworthy at any particular time</td>
</tr>
</tbody>
</table>

Security policies

- Operating systems have to implement policies
  - Mechanisms are used to enforce policies—more on that in a bit
- Different types of security policies
  - Military
  - Commercial
- General goal of security policies: restrict flow of information
Military security

- Information ranked by sensitivity level
  - Lowest: Unclassified
  - Highest: Top Secret
- Information access limited by “need-to-know”
  - Divided into compartments
  - Each piece of information in zero or more compartments
  - Compartments may span levels
- Access to information requires
  - Clearance at the right level
  - Access to the compartment
- More on military security policies
  - Information classified by <rank, compartments>
  - \( S \) dominates \( O \) if and only if
    - \( \text{Rank}(S) \leq \text{Rank}(O) \) and
    - \( S \) has all of the compartments \( O \) has
  - \( S \) can read \( O \) only if \( S \) dominates \( O \)
    - \( S \) needs all compartments, not just “at least one”
    - \( S \) must have a sufficiently high rank as well
  - Two types of sensitivity requirements
    - Hierarchical: rank
    - Nonhierarchical: need-to-know
    - Even a Top Secret clearance isn’t enough to get all information…
Commercial security policies

- Similar to military in many ways
  - Hierarchical ranking
  - Non-hierarchical “compartments”: based on project
    - Not all employees need to know about new products!
- Less formal than military
  - No security clearances (usually)
  - No dominance function (usually)
- Still need to restrict information flow

What about writes?

- So far, mostly about who can read what
  - Ensures confidentiality, but not integrity
- Who can write something?
  - May be able to write something you can’t read!
- Well-formed transactions
  - Maintain consistency between internal data and users’ expectations
- Separation of duty
  - User has the ability to perform all of A, B, and C
  - Different users must perform each to guard against potential for abuse
- Chinese Wall
  - Objects
  - Company groups
  - Conflict classes: only one set of objects (belonging to a single company, typically) can be accessed from each class
Policies are good, but…

- Military and commercial security policies are examples of policies
- What are the underlying models?
- Models can be used to implement particular policies
  - Should be capable of implementing many different types of policies
  - Should be easily implementable in an operating system

Lattice model

- A “lattice” is a mathematical structure
  - One operation defined: \( \leq \)
  - This operation is:
    - Transitive: \( a \leq b \) and \( b \leq c \)
      means that \( a \leq c \)
    - Antisymmetric: \( a \leq b \) and \( b \geq a \)
      means that \( a = b \)
  - Operation \( \leq \) need not be defined for every pair of elements
- Security model based on this approach is the lattice model
Bell-La Padula confidentiality model

- Formalizes military security policy
- Basis for Department of Defense evaluation criteria
- Object $o$ has classification level $C(o)$
- Subject $s$ has classification level $C(s)$
- Read access: $s$ can read $o$ iff $C(o) \leq C(s)$
- Write access: $s$ who can read $o$ can write $p$ iff $C(o) \leq C(p)$
  - Can only write objects to security level at least as high
  - Makes it difficult to declassify objects…
- Information flows up!

Biba integrity model

- Bell-La Padula applies to confidentiality
- Problem: what about integrity?
  - Bell-La Padula lets untrusted subjects write objects to higher confidentiality levels
  - No information leak, but possible integrity issues
- Biba integrity model is the dual of La-Padula
- Ordering is on integrity: $I(s)$ or $I(o)$
  - Write: $s$ can write $o$ only if $I(s) \geq I(o)$
  - Read: if $s$ has read access to $o$, $s$ can have write access to $p$ iff $I(o) \geq I(p)$
- Integrity can only decrease with each successive write
Other security models

- Bell-La Padula and Biba only address reads and writes
- Other models address other operations
  - Creation & deletion
  - Modification of access rights
- Use models to formally prove protection properties
  - Different models can handle different scenarios
  - Formal proof increases belief that system is trustworthy
- Still have to implement the model correctly!

Designing trusted operating systems

- Good security model isn’t enough!
  - Need to design for security
  - Need to implement the design correctly
- Important things to remember
  - Good design is critical!
    - No way to recover if you don’t do this
  - Security considerations have to be central in the OS
    - Adding security as an afterthought is likely to fail
    - Spending more CPU time on security may be necessary (and probably doesn’t hurt these days)
  - Do lots of testing to ensure secure design has been implemented correctly
Design principles (1)

- Least privilege: each user / program should operate with as little privilege as possible to do the operation
  - Example: don’t run as root on your home system, even though you’re probably the sysadmin
  - This is what `sudo` is for
- Economy of mechanism: protection mechanism should be small, simple, and straightforward
  - Complex systems are hard to analyze and verify!
- Open design: security by obscurity doesn’t work
  - Intruders will find hidden things
  - Public scrutiny helps find bugs
- Complete mediation: check every access
  - Don’t assume that previous permissions still hold…

Design principles (2)

- Permission based: default is “access denied”
  - Explicitly allow access to certain objects
- Separation of privilege: access to objects should depend on multiple things
  - Example: require `sudo` access and user password to do root stuff
- Least common mechanism: keep things separate to reduce risk from sharing
- Ease of use: if it’s not easy to use, users will circumvent it
  - Example: nonsense passwords
Security in normal operating systems

- User authentication
- Memory protection
- Access controls for files & I/O devices
- Allocation controls for objects
- Sharing enforcement: require users to share resources
- Guarantee of fair service
- Interprocess communication
- Self-protection: OS must guard its own data

- All of these are necessary for **minimal** security

More security in operating systems

- Separate modules as much as possible
  - In Unix, any module can access data from any other
- Control & check access at each module separately
  - Slows the OS down
  - Makes attacks more difficult!
- Ensure that information doesn’t leak out
  - Scrub resources
  - Keep audit records, just in case
Mandatory (non-discretionary) access control

- **Centralize** access control decisions
  - Owner of an object can’t decide on access to it
  - Access rights granted by higher-level policy
- Example: military security
  - Users can’t determine the access rights for their own objects
- This improves security by reducing the risk that information will be leaked
  - Accidentally
  - Intentionally
- Labeling: attach label to every subject & object
  - Help the system figure out who can access what
  - Labels need to be verifiable…
- Complete mediation: check *every* access!
  - Don’t assume that a success last time still holds

Security-Enhanced Linux (SELinux)

- Implemented by the NSA
  - Not totally secure!
  - However, much better than existing versions of Linux
- Approach: enforce mandatory access control
  - Each process is assigned a domain
  - Each object is assigned a type
- Configuration files describe policies
  - Which domains can access which objects, and how
  - Which domains can interact with one another
  - Allowable domain transitions
- Only applies to “user” processes
  - Doesn’t ameliorate OS bugs
  - Does make “root” safer
SELinux mechanisms

- Basic structure: security context
  - Contains all security attributes for the subject or object
  - Security identifier (SID) maps to security context
- Security manager makes decisions
  - Input is pair of SIDs and desired operation
  - Labeling (transition) decisions: what label should a new object get?
  - Access decisions: should this operation be allowed?
    - May provide an auditing vector as well
- Configuration file may only be edited by one program (enforced by domain / role mechanism)
- Combines type enforcement (TE) and role-based access control (RBAC)
  - TE is fine-grained, but tedious to specify
  - RBAC authorizes users for a set of roles and maps roles to TE domains
- More details in the paper online

Object reuse

- Memory is freed, and then reallocated
- Disk blocks are freed from one file and given to another file
- **Must** ensure that data isn’t left in the resource when it’s freed
  - Overwrite space with a fixed pattern
  - Overwrite space several times with fixed or random pattern
- This can be difficult!
  - Supposedly, disk data can be recovered after 4–6 passes of random data
  - Expensive, but worth it for security reasons
Trusted path

- Intruder may spoof a login screen or other access mechanism
  - How can users detect this?
  - Need a method that’s guaranteed to work!
- Solution: trusted path
  - Set of keystrokes that can’t be “caught” (CTRL-ALT-DEL)
  - Connection that can’t be forged: serial line is the only one on which secure commands can be typed
- This is embedded into the OS
  - No easy way for intruder to change things
  - Hacker has to use trusted path for intrusion -> more difficult

Auditing & accountability

- Record everything!
  - You never know when you might need it
  - Any access is potentially a security hole…
- Audit reduction: cut down on audit data
  - Recording everything gets out of hand quickly
  - Try to cut down on this by summarizing
  - Good: makes problem tractable
  - Bad: some stuff slips through the cracks
Security kernels & reference monitors

- Keep security-related functions in a small part of the kernel: security kernel
  - Unity: all security functions centralized
  - Coverage: all access go through the kernel
  - Verifiability: spend a lot of time making sure the kernel does the right thing
  - Separation: security handled only in the security kernel and not elsewhere
- Reference monitor: portion of the security kernel that actually controls accesses to objects
  - Tamperproof
  - Invoked on every object
  - Very small: can be verified and trusted

Trusted Computing Base (TCB)

- TCB: all of the parts of the trusted OS that we have to trust
  - Security kernel / reference monitor (of course)
  - Processes
  - Memory management
  - Interprocess communication
  - Some files
- TCB must do
  - Process activation
  - Execution domain (protection domain) switching
  - Memory protection
  - I/O operations & protection
Virtualization

- Virtual machine: simulate the real hardware
  - Don’t actually provide direct access to the real hardware!
  - Arbitrate multiple requests coming from multiple virtual machines
- Advantage: virtual machine software guarantees isolation
- Disadvantage: some things (like sharing) are difficult

Structuring OS for secure design

- Layering is a good idea
  - Put more secure-critical functions (as in the TCB) in the innermost layer
  - Layers towards the outside are less trusted
    - Can’t modify structures in inner layers
  - Inner layers can (possibly) modify things in outer layers
- Trust the inner layers more
  - Security kernel fully (?) trusted
  - Outer layers not trusted as much
- Isolation improves security and trust
Why do we need secure file systems?

- Data must be kept secret
  - From outsiders: intruders
  - From insiders: users with insufficient authority
- Data integrity must be preserved
  - Don’t allow unauthorized users to make changes
  - At least allow detection of such changes
- Data must be kept available
  - This is somewhat harder
  - Use aggressive replication
    - Keep lots of copies
    - If any copies are available and unmodified, data is available

Secure file system mechanisms

- Confidentiality
  - Encrypt data at the client
  - Data never exists in the clear except at the user’s computer
- Integrity
  - Use cryptographic hashes
  - Make sure that the hash can only be generated by an authorized user
    - Digital signature
    - HMAC
- Availability
  - Store many copies across many sites (peer-to-peer?)
  - Make it difficult for an attacker to get all of them
Confidentiality

- Keep data secret on a file server
  - Prevent others from seeing it
  - Prevent the file server from seeing the content
  - Sysadmins can be a major source of data leaks!
- Require a password for use
  - Use encryption to protect files
  - Accidental disclosure of encrypted data doesn’t really hurt
  - More difficult to plant fake data if you don’t have a good password
- Restrict plaintext files to user workstation
  - Only trust your local computer
  - Allow server to be run by a third party

Encrypt-on-disk: NCryptFS and others

- Goal: encrypt data on the disk itself
  - Typically a local file system
  - Provide password to use the files
- Files are kept encrypted as long as possible
  - Encrypted in the cache
  - Decrypted when copied to user space
- Data encrypted one block at a time
  - Keys done on a per-file basis
  - Files may share keys
So what’s the big deal?

- Encrypt-on-disk sounds easy!
- There are plenty of issues to deal with!

- Doing it locally is easy, but networks make things harder
  - How are keys transported?
- Sharing makes things harder
  - How can different users share keys easily?
    - A user may be in multiple “groups”
  - How can different users share the same block in the (encrypted) cache?
- What about metadata?
  - Are file names / sizes / owners / etc. encrypted?
  - How does backup work if this information isn’t available?
- Security issues
  - No keys stored on disk!
  - Keys not stored in swap, hopefully

Many similar systems

- CFS
  - One of the first cryptographic file systems
  - Directory structure looked like a big (featureless) file
- TCFS
  - Encrypts on a file-by-file basis
  - Works over a network
- EFS (Windows Encrypting File System)
  - Not very strong…
- BestCrypt
  - Encrypting block device driver
  - Loopback device: can be used with any device, not just disk
- NCryptFS
  - Performs well: high speed
  - Encrypts file-by-file
  - Stackable with other file system operations (like compression)
What about integrity?

• How much do you trust your file system?
  ◆ Someone could break in and modify files
  ◆ How can you find out?
• Network file systems are worse!
  ◆ Problems can occur at the disk or in the network
  ◆ “Spoofed” servers can be an issue
  ◆ This can be a problem for software distribution
• How do you know the remote file server is valid?
• How can you set up valid mirrors?
  ◆ How can you tell the mirror is valid?

Solution: Secure File System (SFS)

• Use hashing for security
  ◆ Tree of hashes
  ◆ Digitally sign the root for security
• Include the signature in the pathname
  ◆ Soft link that includes the long signature
  ◆ Check the signature each time you access a file
• Anyone can mirror the tree
  ◆ Root of the tree is signed by the creator
  ◆ Signature can be checked by anyone
• Problem: this is pretty much read-only
  ◆ Still good for software distribution
  ◆ Read-only may be the way to go in many cases
Hash trees

- Goal: sign a bunch of stuff with just one signature
  - Faster
  - Fewer signatures to carry around
- Keep a tree of hashes
  - Hash each block (or file)
  - Make lists of hashes
  - Hash them, too
- If hash is truly cryptographic, this works!
  - List of hashes has a unique hash value
  - “Hard” to find another block that matches…

Secure network file systems

- Storage Service Providers
  - Sell storage on a network for a fee
  - May be totally untrustworthy
    - You can get your money back, but…
    - Hard to get your data back if it’s leaked
- Goal: provide a service so that
  - Storage is private: unauthorized users can’t see anything
  - Storage is authentic: no fake data
- Stealing the disk doesn’t do you any good!
Goal: Secure Distributed File System

- Distributed file systems have major security issues
  - Users want to read & write data easily
  - Users want to share their data with others (arbitrarily)
  - Users have no control over disk security
    - Backups have readable data on them!
    - How much do you trust your hardware vendor / sysadmin / storage service provider?
- Address these problems by
  - Encrypting all data at the client
  - Allowing arbitrary sharing of data
  - Allowing clients to verify the integrity of written data
    - Disks don’t have sufficient information to decrypt the data stored on them!

Design Principles

- For each file block on disk, keep sufficient information to
  - Decode the data (at the client)
  - Validate the sender of the data
  - Ensure data integrity
- Use encryption to keep data secret on disk and in transit
  - Decryption only occurs at the client
  - Sufficient information to decrypt only available at client!
- Allow anyone to read a block!
  - Can still use access controls, but not a problem if a block “leaks out”
- Restrict writing to authorized users
- Prevent compromise of data
  - Impossible to protect against denial of service
  - Loss of data may occur => make sure it’s noticed!
- Three similar security schemes
  - Trade off resistance to intrusion for speed
High-level System Design

- File objects
  - Decoding information kept as a pointer to a key object
  - All blocks in a file encrypted with same key
  - Metadata kept as in a regular file system

- Key objects
  - Store keys to decrypt files
  - Single key object can be used by multiple files

- Certificate object
  - Verifies writers: prevents data loss due to overwriting
  - Modification necessary only when new users added

Key Objects

- Key objects hold keys for encrypting & decrypting files
  - One key object per “unique” group
  - Many files can use one key object

- Key objects contain
  - File encryption key
    - Encrypted with each user’s / group’s public key
    - Disk doesn’t have enough information to decrypt key
  - Permission bits for modifying the key object

- On file open, appropriate line of key object sent to user

<table>
<thead>
<tr>
<th>KeyFileID</th>
<th>Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserID_0</td>
<td>E(Pub,Key)</td>
</tr>
<tr>
<td>UserID_1</td>
<td>E(Pub,Key)</td>
</tr>
<tr>
<td>UserID_2</td>
<td>E(Pub,Key)</td>
</tr>
<tr>
<td>UserID_3</td>
<td>E(Pub,Key)</td>
</tr>
</tbody>
</table>

Encrypt with key

Client

To disk

File block
Where do users’ private keys come from?

- Key objects require a user’s private key to decrypt file keys
- Private keys can be obtained by “standard” mechanisms such as
  - Function of user’s password
  - Central key server (as in Kerberos)
  - Smart card
- Combine above mechanisms for added security
- Obtaining private keys is a problem for any security system

Certificate Objects

- One certificate object per server
- One entry per user / group
  - User ID
  - Public key
    - Convenience for creating key objects
    - Future use for stronger security schemes
  - HMAC (keyed hash) key
  - Timestamp: prevent replay attacks
    - Kept per-user to avoid problems of clock skew
    - May use other mechanisms to prevent replay attacks
- Certificate object written securely

<table>
<thead>
<tr>
<th>CertificateFileID</th>
<th>Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserID_0 Pub_0</td>
<td>HMAC_KEY_0</td>
</tr>
<tr>
<td>UserID_1 Pub_1</td>
<td>HMAC_KEY_1</td>
</tr>
<tr>
<td>UserID_2 Pub_2</td>
<td>HMAC_KEY_2</td>
</tr>
<tr>
<td>UserID_3 Pub_3</td>
<td>HMAC_KEY_3</td>
</tr>
</tbody>
</table>
Data Objects (File Blocks)

- Security information varies with each scheme
  - Schemes 1 & 2: signed cryptographic checksum
  - Scheme 3: HMAC (not stored on disk)
- Headers need not be stored contiguous to data on disk
- Data and checksum are encrypted
  - Client can verify integrity
  - Impossible for intruder to modify data & recalculate
- Data exists in clear only at the client!

Security Scheme 1: Write

- Data write
  - Client encrypts data with $K_B$ – $K_B$ obtained from key object
  - Client signs cryptographic hash
  - Server verifies signature
  - Server stores signed hash and data
  - Server denies writes to clients without permission
- Problem: server must verify signed hash
  - This is slow (~5-10 ms!)
  - Increases latency for writes
- Problem: client must generate signed hash
  - This is also slow!
- How can server load be reduced?
Security Scheme 1: Read

- Data read
  - Server returns block with signed cryptographic hash
  - Client verifies hash
  - Client decrypts data with $K_B$
    - $K_B$ obtained from key object

- Problem: client must verify signed hash
  - This is slow (~5-10 ms)!
  - Increases latency for reads

- Scheme 1 is secure, but at what cost?

Security Scheme 2

- Identical to Scheme 1, except
  - Client also generates HMAC to authenticate to disk
  - Disk only checks HMAC

- Client still relatively slow
  - Signature generated on write
  - Signature verified on read

- Server load reduced
  - HMAC only requires hashing: very fast!
  - One server can support more clients
**Security Scheme 3: Write**

- **Data write**
  - Client encrypts data with $K_B$
  - Client generates HMAC using client/server shared secret
  - Server verifies HMAC
  - Server denies writes to clients without permission
- **No more public-key operations!**
  - No ability to identify the last writer of a block
  - Intruder can read certificate object off disk and write data illegally using HMAC
    - If intruder has raw disk access, he can just write the data!
  - No undetectable fakes
    - Hash over plaintext detects writers who don’t have $K_B$
  - Much faster!

**Security Scheme 3: Read**

- **Data read**
  - Server generates new HMAC for the client reading the data
  - Server returns block with HMAC
  - Client verifies HMAC
  - Client decrypts data
  - HMAC operations can be omitted by decrypting and checking the plaintext hash
- **Client doesn’t do public-key operations**
- **Much faster**
  - Cryptographic hashes
  - Symmetric encryption
  - Both run at 10+ MB/s
Read Performance

- Schemes 1 & 2
  - Reads limited by client signature verification
  - OK for random reads
- Scheme 3
  - Little performance loss on random reads of any size
  - 1.5 MB/s drop in throughput on sequential reads
    - Could be hidden by pipelining requests
  - Disk CPU not heavily loaded (more clients => more bandwidth)
- All schemes have reasonable sequential performance

Write Performance

- Schemes 1 & 2 require signature generation
  - Very slow!
  - Limit performance to 1.5 MB/sec
  - Could be more with larger blocks…
- Scheme 3 requires only hashing
  - Random blocks written at full speed
  - Sequential blocks show a 1 MB/s slowdown
  - Disk CPU not loaded heavily (more clients => more bandwidth)
Security Issues

- Data has end-to-end protection
  - HMAC protects data in transmission
  - Use of HMAC & timestamp prevents spoofing
  - Stealing HMAC == physical access to disk
    - Write fake data blocks on disk, but...
    - Client will easily tell fakes by checking cryptographic hash over plaintext (both hash and plaintext are encrypted)

- Data on disk can’t be read
  - All data stored encrypted
  - Decryption keys not stored in the clear on the disk

- Denial of service still possible
  - Flood disks with read (or write) requests
  - Attack the disks with a sledgehammer...
  - Protect data by using copy-on-write (similar to self-securing storage)

Creating a file

- Creating a file using an existing key object
  - New file simply points to the existing key object
  - All permissions & users listed in key object can use the new file

- Creating a file using a new key object
  - Client sends contents of key object to disk
    - Client needs only public keys of all users to be included in the new key object
    - Client generates a new RC5 key and encrypts it with the public keys of all users in the key object
  - File creation proceeds as above
Is SNAD enough?

- Problems still exist with
  - Users’ access to files changing: must reencrypt files
  - “Freshness” of data

- Freshness is a particular problem!
  - File system accepts reads and writes from different users
  - Transfers are real, but user A doesn’t see changes made by user B, and vice versa
  - Particularly a problem if A and B are trying to collaborate

- Solution uses hash trees

SUNDR

- Build a hash tree of the entire file system
- Update it every time data is written
- Return the (signed) root of the tree with each file access (or periodically)
- Ensures that, if data changes, you have to see the freshest data
- File server can still refuse to propagate changes to users!
  - Essentially the same as maintaining two separate servers
  - If A’s writes are seen by B, all of B’s writes to that point must also be visible
- This eliminates a potential problem with untrusted servers
- May be somewhat slow: digital signatures aren’t that fast…
Plutus

- Untrusted servers
  - Data encrypted at the client
- Lazy revocation
  - Access to a file isn’t revoked until the file is changed
  - Revoked reader can’t read updated files
- Uses file groups in a way similar to SNAD
- New concept: key rotation
  - Use public key encryption to derive new key for lockbox
    - Owner generates new key from old key using private key
    - Anyone can generate older keys from newer keys
  - Similar technique used to sign contents of files
- Good authentication and revocation mechanisms, but can be slow because of public key usage