Security Policies and Enforcement Mechanisms

Terminology and concepts

- **Principals, Subjects, Objects**
- **Principle of least privilege**
  - Throughout execution, each subject should be given the minimal access necessary to accomplish its task
    - Needs mechanisms for rights amplification and attenuation
- **Reference monitors**
  - Abstract machine that mediates all access
- **Security kernel**
  - Hardware, firmware and software elements that implement the reference monitor
- **Trusted Computing Base**
  - Totality of protection mechanisms in the system
  - Smaller TCB => Greater assurance that the system is secure

Overview

- **Access control**
  - Mandatory Vs Discretionary policies
- **Capabilities**
- **Information flow**
- **Least privilege principle**
  - Domain and type enforcement (DTE)
  - POSIX Capabilities
- **Other policies**
  - Chinese wall
  - Clark-Wilson
- **Policies for containing untrusted code**
- **Manageability**
  - Role-based access control (RBAC)
  - Deletation and trust management

Access control

- **Typically, three kinds of entities**
  - User (principal)
  - Subject: typically, a process acting on behalf of user
  - Object: files, network sockets, devices, …
- **Goal: Control access to *operations* performed by *subjects* on *objects***
  - Examples of operations
    - Read
    - Write
    - Append
    - Execute
    - Delete
    - Change permission
    - Change ownership

Discretionary Access Control

- **Permissions specified by users**
  - permission on an object is set by its owner
  - typical on most OSes (UNIX, Windows, …)
- **Represented using a matrix**
  - Indexes by subject and object
  - Each element specifies the rights available to subject on that object (read, write, etc.)
  - Implementations
    - ACL (associated with an object, represents a column)
    - Capabilities (associated with subject, represents a row)
- **Improve manageability using indirection**
  - Groups
  - Roles (RBAC)
  - Inheritance
  - Negative permissions

Implementation of DAC on UNIX

- **All resources are “files”**
- **Each file has a owner and group owner**
- **Permissions divided into 3 parts**
  - For owner, group, and everybody else
  - 3 bits per part: read/write/execute
- **Subjects inherit the userid of parent**
  - Programs that perform user authentication need to set this info
  - Exception: setuid programs (privilege delegation/amplification mechanism)
    - Suid and sgid bits
- **No permission checks on superuser (userid 0)**
  - Permission checks based on userid --- usernames used only for login
- **Defaults (umask)**
- **Changing permission**
- **Changing ownership**
- **Recent additions**
  - Access control lists
  - Sticky bit

Effective, Real and Saved UID/GID

- **Effective: the uid used for determining access privileges**
- **Real: the “real” user that is logged on, and on whose behalf a process is running**
- **Saved: allows processes to temporarily relinquish privileges but then restore original privileges**
  - When executing a setuid executable, original euid is saved (or it could be explicitly saved)
  - Setting userid to saved userid is permitted

DAC on Windows Vs UNIX

- **OO-design: permissions can differ, depending on type of object**
  - NTFS files offer additional rights: delete, modify ACL, take ownership
    - Files inherit permission from directory
  - Use of Registry for configuration data
    - Richer set of access permissions for registry entries (e.g., enumerate, create subkey, notify, …)
- **Mandatory file system locks**
- **No setuid mechanism**

Capabilities

- **“Tickets:” subject presents capabilities to the resource to gain access**
  - Must be unforgeable
  - Transferable
- **Examples**
  - File descriptors
  - Passwords

Capabilities

- **Not widely used in OSes**
  - More difficult to implement than ACLs
    - Need forever unique object ids that don't change
    - Need to use crypto or rely on OS primitives that may be hard to realize
  - Difficult to manage
    - How do we determine the permissions held by a user?
    - Do we want to allow them to pass around their capability? What about theft?
    - How long do we store them?
    - How can we revoke permissions?
- **Provide a better framework than ACLs when policy enforcement is NOT centralized**
  - Kerberos uses capabilities for access across hosts
    - Uses capabilities with different time scales
    - Accesses within a host typically based on ACL mechanism of host OS
  - Web applications use cookies containing sessionids to indicate when a user has successfully authenticated

Mandatory Access Control (MAC)

- **DAC Limitations**
  - provides no protection if a resource owner did not bother to set the ACL properly
  - assumes that users are in full control of programs
    - What if a program changes permissions without user's knowledge?
    - In general, "Trojan horse" programs can subvert DAC
- **To overcome these problems, MAC moves the responsibility to a central point, typically the system administrator**
  - Organizations want to control access to their resources
  - Don't want to rely on individual employees to ensure that organizational policies are enforced

MAC Example: MLS

- **Motivation: DAC does not provide any way to control the manner in which information is used – it only says whether it can be accessed or not.**
- **MLS policies control information flow, and hence control how information is used**
- **Developed originally in the context of protecting secrets in the military**

MLS: Confidentiality Policies

- **Objects are labeled with a level**
  - Labels correspond to points in a lattice
  - Typical levels used in military include:
    - unclassified, classified, secret, top secret
- **Subjects associated with clearance levels**
  - A subject can access an object is his clearance level is equal to or above the object's level
- **Information is also compartmentalized**
  - "Need-to-know" principle is used to decide who gets to access what information
    - e.g., top-secret information regarding nuclear fuel processing is made available to those working on nuclear-related projects

MLS: Bell-LaPadula Model [1973]

- **To ensure that sensitive information does not leak, we need to ensure:**
  - No “read-up:”
    - A subject S can read object O only if C[S] >= L[O]
  - No “write-down:”
    - A subject can write an object O only if C[S] <= L[O]
    - Prevents indirect flows where a top-secret-clearance subject reads a top-secret file and writes to a secret file, which may then be read by someone with a lower (ie secret) clearance
  - Based on the idea that any subject that reads information at a certain level has the potential to leak information at that level whenever it outputs anything.

MLS: Biba Model (Integrity)

- **Designed to ensure integrity rather than confidentiality**
  - In non-military settings, integrity is more important
- **Conditions**
  - No “read-down:”
    - A subject S can read object O only if C[S] <= L[O]
    - A subject’s integrity can be compromised by reading lower integrity data, so this is disallowed
  - No “write-up:”
    - A subject can write an object O only if C[S] >= L[O]
    - The integrity of a subject’s output can’t be greater than that of the subject itself.
- **Variation: Low Water-Mark Policy (LOMAC)**
  - Allow read-downs, but downgrade subject to the level of object
- **Both policies ensure system integrity**

Problems with Information Flow

- **In a nutshell: difficult to set up/use**
  - "Label creep:" More and more objects become sensitive, making it difficult for the system to be used by lower-clearance subjects
  - Exceptions need to be made, e.g., an encryption programs
    - "Trusted" programs are allowed to be exempted from "*"-property
    - But exceptions are misused widely, since it is hard to configure whole systems carefully so that "*"-property can be enforced without breaking functionality
- **Motivate alternate approaches, or hybrid approaches**

Alternative Approaches

- **Key goal: Mitigate damage that may result from all-powerful root privileges**
  - Break down root privilege into a number of sub-privileges
  - Decouple user privileges from program privileges
- **Examples**
  - Domain and type enforcement
    - SELinux
  - “Linux capabilities”
    - not to be confused with capabilities as described earlier

Domain and Type Enforcement

- **Subjects belong to domains**
  - Users have default domains, but not all their processes belong to the same domain
    - Some processes transition to another domain, typically when executing another program
- **Objects belong to types**
- **Policies specify**
  - Which domains have what access rights on which types
  - Domain transitions
- **Domain transitions are an important feature**
  - Enable application of least-privilege principle
  - Example: a media player may need to write its configuration or data files, but not libraries or config files of other applications

DTE and SELinux

- **Security-enhanced Linux combines standard UNIX DAC with DTE**
- **Intuitively, the idea is to make access rights a function of (user, program, object)**
- **Roughly speaking, MLS requires us to trust a program (and not enforce "*"-property), or fully trust it (ie it may do whatever it wants with information that it read)**
  - In contrast, DTE allows us to express limited trust, i.e., grant a program only those rights that it needs to carry out its function

DTE/SELinux Vs Information Flow

- **In practice DTE has turned out to be "one policy per application"**
  - Scalability is clearly an issue
  - In addition, SELinux policies are quite complex
  - While DTE is able to gain additional power because it captures the fact that trust is not transitive, this very feature makes DTE policies difficult to manage
    - What overall system-wide assurances can be obtained, given a set of DTE policies developed independent of each other
- **In contrast, information flow policies are simple, easier to understand, and more closely relate to higher level objectives**
  - Confidentiality or Integrity

Linux (POSIX) Capabilities

- **Decompose root privilege into a number of “capabilities”**
    - CAP_CHOWN
    - CAP_DAC_OVERRIDE
    - CAP_NET_BIND_SERVICE
    - CAP_SETUID
    - CAP_SYS_MODULE
    - CAP_SYS_PTRACE
    - ...
- **Effective, Permitted and Inheritable capabilities**
    - Effective: accesses will be checked against this set
    - Permitted: superset of effective, cannot be increased
        - Effective set can be set to include any subset of permitted
    - Inheritable: capabilities retained after execve
        - at execve, permitted and effective sets are masked with inheritable
- **attaching capabilities to executables**
    - Allowed: capabilities not in this set are taken away on execve
    - Forced: “setuid” like feature --- given to executable even if parent does not have the capability
    - Effective: Indicates which of the permitted bits are to be transferred to effective

Commercial Policies

- **High-level policies in commercial environments are somewhat different from those suitable for military environments**
- **Examples**
  - Chinese Wall (conflict of interest)
  - Clark-Wilson
- **Common principles**
  - Separation of duty: critical functions need to be performed by multiple users
  - Auditing: ensure actions can be traced and attributed, and if necessary, reverted (recoverability)

Clark-Wilson Policy

- **Focuses on data integrity rather than confidentiality**
  - Based on the observation that in the “real-world,” errors and fraud are associated with loss of data integrity
- **Based on the concept of well-formed transactions**
  - Data is processed by a series of WFTs
  - Each WFT takes the system from one consistent state to another
    - Operations within a WFT may temporarily make system state inconsistent
  - While the use of WFTs guarantee consistency of system state, we need other mechanisms to ensure integrity of WFTs themselves
    - Was that a fraudulent money transfer? Was that travel voucher properly inspected?
      - Relies primarily on separation of duty
  - Auditing to verify integrity of transactions
  - Maintain adequate logs so that WFTs in error can be undone

Chinese Wall Policy

- **Addresses “conflict of interest”**
  - Common in the context of financial industry
  - Regulatory compliance, auditing, advising, consulting,..
- **Defined in terms of**
  - CD: objects related to a single company
  - COI classes: sets of companies that are competitors
  - Policy: no person can have access to two CDs in the same COI class
    - Implies past, present or future access

Policies and Mechanisms for Untrusted Code

- **Isolation**
  - Two-way isolation
    - Chroot jails
    - Userid-based isolation
    - Virtual machines
  - One-way isolation
    - Read access permitted, but write access denied
- **System-call sandboxing**
  - Linux seccomp and seccomp-bpf
  - Delegation
- **Information flow**

chroot jails

- **Makes the specified directory to be the root**
  - Process (and its children) can no longer access files outside this directory
- **Requires root privilege to chroot**
  - For security, relinquish root privilege after chroot
  - All programs, libraries, configuration and data files used by this process should be within this chroot'ed dir
- **Isolation limited to file system**
    - e.g., it does not block interprocess interactions
  - For this reason, chroot jail is useful mainly to limit privilege escalation; but the mechanisms is insecure against malicious code.

Userid based isolation

- **Create a new userid for running untrusted code**
  - Real user's userid is not used, so the "Trojan horse" problem of altering permissions on user's files is avoided
- **Android uses one userid for each app**
  - Default permissions are set so that each app can read and write only the files it owns (except a few system directories)
- **Protects against malicious interprocess interactions**
  - kill, ptrace, …
- **Better than chroot, but still insufficient against malicious code**
  - Can subvert benign processes by creating malicious files that may be accidentally consumed by them
    - Many sandbox escape techniques work this way
  - Too much information available via /proc, as well as system directories that are public: Can use this info to exploit benign processes via IPC

One-way isolation

- **Full isolation impacts usability**
  - untrusted applications are unable to access user's files
  - makes it difficult to use nonmalicious untrusted applications
- **One-way isolation**
  - Untrusted application can read any data, but writes are limited
    - cannot overwrite user files
    - More importantly, benign applications don't ever see untrusted files
      - Eliminates the possibility of accidental compromise
- **Key issues:**
  - Ensuring consistent view
    - Application creates a file and then reads it, or lists the directory
    - Inconsistencies typically lead to application failures
  - Failures due to denied write permission
    - Can overcome by creating a private copy of the file
- **Both issues overcome using copy-on-write file system**
- **Note: does not protect against lost of confidential data**
  - Needs additional policies (which files should be unreadable for untrusted code)
- **Note: securing user interactions is always a challenge, especially because of how X-windows is designed**

System-call sandboxing: seccomp

- **Seccomp is a Linux mechanisms for limiting system calls that can be made by a process**
  - Processes in the seccomp sandbox can be make very few system calls (exit, sigreturn, read, write).
- **More secure than previous mechanisms, but greatly limits actions that can be performed by a sandboxed process**
  - Useful if setup properly, e.g., in Chrome and Docker
- **Seccomp-bpf is a more recent version that permits configurable policies**
  - Allowable syscalls specified in the Berkeley packet filter language
  - Policies can reference syscall name and arguments in registers
- **Unfortunately, most interesting policies are out-of-scope, as they reference data in process memory, e.g., file names**
  - For this reason, seccomp-bpf is not much more useful than seccomp

System-call delegation

- **Used in conjunction with strict syscall sandboxing**
  - Key idea: Delegate dangerous system calls to a helper process
  - Helper process is trusted
    - it cannot be manipulated by untrusted process
    - can implement arbitrary, application-specific access control logic
    - avoids race conditions
- **Works only if**
  - System call semantics permits delegation
    - e.g., not applicable fork or execve
  - Results can be transferred back transparently to untrusted process
    - e.g., file descriptors can be sent over UNIX domain sockets using sendmsg

Securing untrusted code using information flow

- **Untrusted code = low integrity, benign code = high integrity**
- **Enforce the usual information flow policy that**
  - Deny low integrity subject's writes to high integrity objects
    - Prevents "active subversion"
  - Deny high integrity subject's read of low integrity objects
    - Prevents "passive subversion"
      - fooling a user (or a benign application) to perform an action, e.g., click an icon on desktop
      - exploit a benign process, e.g, benign image viewer compromised by reading a malicious image file
- **Can provide strong guarantee of integrity**
  - Not subject to "sandbox escapes"
- **Usability issues still need to be addressed**

Policy Management

- **Goal: simplify the set up and administration of security policies**
- **Topics**
  - Role-based access control (RBAC)
  - Administrative policies
    - Who can change what policies
  - Delegation and trust management

RBAC

- **Roles vs groups: Very closely related concepts, but we can make a distinction**
  - Role: a set of permissions
  - Group: a set of users
- **Roles and groups provide a level of indirection that simplifies policy management**
  - Based on the functions performed by a user, he/she is given one or more roles
    - When the user's responsibilities change, the user's roles are updated
    - When the permissions needed to perform a function are changed, the corresponding role's permissions are updated
      - Does not require any updating of user information

Delegation

- **Ability to transfer certain rights to another entity so that it may act on behalf of the first entity**
- **Delegation is necessary for managing authorizations in a distributed system**
  - Decentralized/distributed control
- **How to implement delegation**
  - The issue is one of trust and granularity
  - Multiple levels of delegation rely on a chain of trust
    - Can be implemented using certificates
- **Trust management**
  - Systems designed to manage delegation, and enforce security policies in the presence of delegation rules and certificates