Computer Security CS 426

Lecture 28

SELinux & UMIP

Security Enhanced Linux (SELinux)

- Developed by National Security Agency (NSA) and Secure Computing Corporation (SCC) to promote MAC technologies
- MAC functionality is provided through the FLASK architecture
- Policies based on type-enforcement model
- Integrated into 2.6 kernels
- Available in many Linux distributions (e.g., Fedora, Redhat Enterprise, Debian, Ubuntu, Hardened Gentoo, openSUSE, etc.

FLASK

- **Fl**ux **A**dvanced **S**ecurity **K**ernel
- Developed over the years (since 1992) in several projects: DTMach, DTOS, Fluke
- General MAC architecture
- Supports flexible security policies, “user friendly” security language (syntax)
- Separates policies from enforcement
- Enables using more information when making access control decisions
  - E.g., User ids, Domains/Types, Roles

Type Enforcement (or Domain Type Enforcement)

- Type enforcement first proposed by W. E. Boebert and R. Y. Kain.
  - A Practical Alternative to Hierarchical Integrity Policies. In In *Proceedings of the 8 National Computer Security Conference*, 1985.
  - Aim at ensuring integrity
- Key Idea for Type Enforcement:
  - **Use the binary being executed to determine access.**
  - What do DAC and MAC use?

Rationale of Type Enforcement (1)

- Integrity level should be associated with programs (rather than processes)
  - Trust in programs is required for integrity
- Examples of assured pipelines:
  - **Labeling**: All printouts of documents must have security labels corrected printed by a labeller.
  - **Encrypting:** Before sending certain data to an output channel, it must be encrypted by an encryption module
- Data must pass certain transforming system before going to certain outputs

Rationale of Type Enforcement (2)

- To ensure assured pipelines are implemented correctly, needs to show
  - Transforming subsystems cannot be bypassed
  - Transformations cannot be undone
    - This and above are global properties, must be enforced by access control policies
  - Transformations must be correct
    - Use program proofing techniques

Rationale of Type Enforcement (3)

- For the labeling example, want to ensure
  1. Only the labeler module produces labeled data
  2. Labeled data cannot be modified
  3. Output module accepts labeled data only
- What integrity levels to use for labeled & unlabeled data?
  – Only reasonable choice is to labeled data have higher integrity
  – Implies: the labeling module must be trusted

Domain-type Enforcement: High-level Idea

- Add a new access matrix
  - One row for each subject domain (more or less)
  - One column for each pair (object type, security class)
  - Each cell contains all operations the subject can perform on objects of a particular type and security class

Domain-type Enforcement (1)

- Each object is labeled by a type
  - Object semantics
  - Example:
    - /etc/shadow etc_t
    - /etc/rc.d/init.d/httpd httpd_script_exec_t
- Objects are grouped by object security classes
  - Such as files, sockets, IPC channels, capabilities
  - The security class determines what operations can be performed on the object
- Each subject (process) is associated with a domain
  - E.g., httpd_t, sshd_t, sendmail_t

Domain-type Enforcement (2)

- Access control decision
  - When a process wants to access an object
  - Considers the following: process domain, object type, object security class, operation

- Example: access vector rules
  - allow sshd_t sshd_exec_t: file { read execute entrypoint }
  - allow sshd_t sshd_tmp_t: file { create read write getattr setattr link unlink rename }

Limitations of the Type Enforcement Model

- Result in very large policies
  - Hundreds of thousands of rules for Linux
  - Difficult to understood
- Using only programs, but not information flow tracking cannot protect against certain attacks
  - Consider for example: httpd -> shell -> load kernel module

SELinux in Practice

- Theoretically, can be configured to provide high security.
- In practice, mostly used to confine daemons like web servers
  - They have more clearly defined data access and activity rights.
  - They are often targets of attacks
  - A confined daemon that becomes compromised is thus limited in the harm it can do.
- Ordinary user processes often run in the unconfined domain
  - not restricted by SELinux, but still restricted by the classic Linux access rights.

UMIP

- Usable Mandatory Integrity Protection for Operating Systems
  - Ninghui Li, Ziqing Mao, and Hong Chen
    In *IEEE Symposium on Security and Privacy*, May 2007.

Motivation

- Host compromise by network-based attacks is the root cause of many serious security problems
  – Worm, Botnet, DDoS, Phishing, Spamming
- Why hosts can be easily compromised
  – Programs contain exploitable bugs
  – The discretionary access control mechanism in the operating systems was not designed to take buggy software in mind

Six design principles for usable access control systems <1>

- *Principle 1: Provide "good enough" security with a high level of usability; rather than "better" security with a low level of usability*
  - Need to trade off "theoretical security" for usability
- *Principle 2: Provide policy, not just mechanism*
  - Go against the UNIX "mechanism-but-not-policy" philosophy
- *Principle 3: Have a well-defined security objective*
  - Simplify policy specification while achieving the objective

Six design principles for usable access control systems <2>

- *Principle 4: Carefully design ways to support exceptions in the policy model*
  - Design exception mechanisms to the global MAC policy rules to minimize attack surface
- *Principle 5: Rather than trying to achieve “strict least privilege”, aim for “good-enough least privilege”*
  - Aim also at minimizing policy specifications
- *Principle 6: Use familiar abstractions in policy specification interface*
  - Design for psychological acceptability

The UMIP Model: Security Objective

- Protect against network-based attacks
  - Network servers and client programs contain bugs
  - Users may make careless mistakes, e.g., downloading malicious software and running them
  - Attacker does not have physical access to the host
- The security property we want to achieve
  - The attacker cannot compromise the system integrity (except through limited channels)
    - E.g, install a RootKit, gain the root privileges
  - The attacker can get limited privileges
    - Run some code
  - After a reboot, the attacker does not present any more

The UMIP Model: Usability Objectives

- Easy policy configuration and deployment
- Understandable policy specification
- Nonintrusive: existing applications and common usage practices can still be used

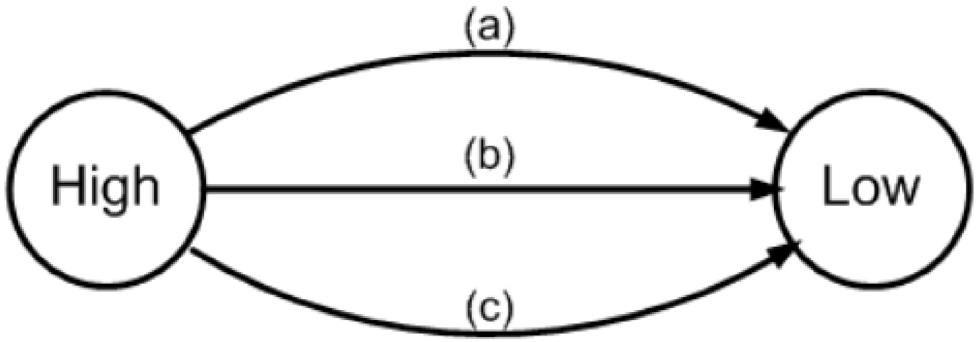
Basic UMIP Model

- Each process is associated with one bit to denote its integrity level, either high or low
  - A process having low integrity level might have been contaminated
- A low-integrity process by default cannot perform any sensitive operations that may compromise the system
- Three questions
  - How to do process integrity tracking?
  - What are sensitive operations?
  - What kinds of exceptions do we need?

Process Integrity Tracking

- Based on information flow

When a process is created, it inherits the parent's IL



The state-transition rules for processes:
(a): receive remote network traffic
(b): receive IPC traffic from a low-integrity process
(c): read a low-integrity file

File Integrity Tracking

- Non-directory files have integrity tracking
  - use the sticky bit to track whether a file has been contaminated by a low-integrity process
  - a file is low integrity if either it is not write-protected, or its sticky bit is set
  - the sticky bit can be reset by running a special utility program in high integrity
    - allow downloading and installing new programs

Sensitive Operations: Capabilities

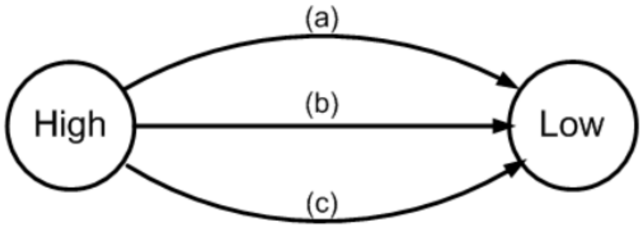
- Non-file sensitive operations
  - E.g., loading a kernel module, administration of IP firewall,…
- Using the Capability system
  - Break the root privileges down to smaller pieces
  - In Linux Kernel 2.6.11, 31 different capabilities
- Identify each capability as one kind of non-file sensitive operation

Sensitive Operations: File Access

- Asking users to label all files is a labor intensive and error-prone process
- Our Approach: Use DAC information to identify sensitive files
- Read-protected files
  - Owned by system accounts and not readable by world
  - E.g., /etc/shadow
- Write-protected files
  - Not writable by world
  - Including files owned by non-system accounts

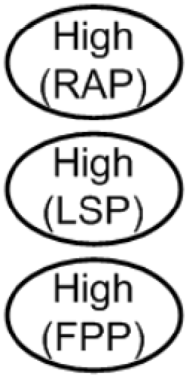
Exception Policies: Process Integrity Tracking

- Default policy for process integrity tracking



The state-transition rules for processes:
(a): receive remote network traffic
(b): receive IPC traffic from a low-integrity process
(c): read a low-integrity file

- Exceptions:



: maintain the integrity when (a) happens
: maintain the integrity when (b) happens
: maintain the integrity when (c) happens

- Examples
  - RAP programs: SSH Daemon
  - LSP programs: X server, desktop manager

Exception Policies: Low-integrity Processes Performing Sensitive Operations

- Some low-integrity processes need to perform sensitive operations normally
- Exception:

: can do operations allowed by special privileges

- Examples:
  - FTP Daemon Program: /usr/sbin/vsftpd
  - Use capabilities: CAP_NET_BIND_SERVICE, CAP_SYS_SETUID, CAP_SYS_SETGID, CAP_SYS_CHROOT
  - Read read-protected files: /etc/shadow
  - Write write-protected files: /etc/vsftpd, /var/log/xferlog

Implementation & Performance

- Implemented using Linux Security Module
  - no change to Linux file system
- Performance
  - Use the Lmbench 3 and the Unixbench 4.1 benchmarks
  - Overheads are less than 5% for most benchmark results

Part of the Sample Policy

Services and Path of the Binary	Type	File Exceptions	Capability Exceptions
SSH Daemon /usr/sbin/sshd	RAP		
Automated Update: /usr/bin/yum	RAP		
/usr/bin/vim	FPP		
/usr/bin/cat	FPP		
FTP Server /usr/sbin/vsftpd	NONE	(/var/log/xferlog, full) (/etc/vsftpd, full, R) (/etc/shadow, read)	CAP_SYS_CHROOT CAP_SYS_SETUID CAP_SYS_SETGID CAP_NET_BIND_SERVICE
Web Server /usr/sbin/httpd	NONE	(/var/log/httpd, full, R) (/etc/pki/tls, read, R) (/var/run/httpd.pid, full)	
Samba Server /usr/sbin/smbd	NONE	(/var/cache/samba, full, R) (/etc/samba, full, R) (/var/log/samba, full, R) (/var/run/smbd.pid, full)	CAP_SYS_RESOURCE CAP_SYS_SETUID CAP_SYS_SETGID CAP_NET_BIND_SERVICE CAP_DAC_OVERRIDE
NetBIOS name server /usr/sbin/nmbd	NONE	(/var/log/samba, full, R) (/var/cache/samba, full, R)	
Version control server /usr/bin/svnserve	NONE	(/usr/local/svn, full, R)	

Differences with Other Integrity Models

- Use multiple policies from the Biba model
  - subject low water for most subjects/processes
  - ring policy for some trusted subjects
    - e.g., ssh daemon, automatic update programs
  - object low water for some objects
- Each object has a separate protection level and integrity level
  - integrity level for quality information
  - protection level for important
    - read protection level inferred from DAC permissions on read
    - write protection level inferred from DAC permissions on write

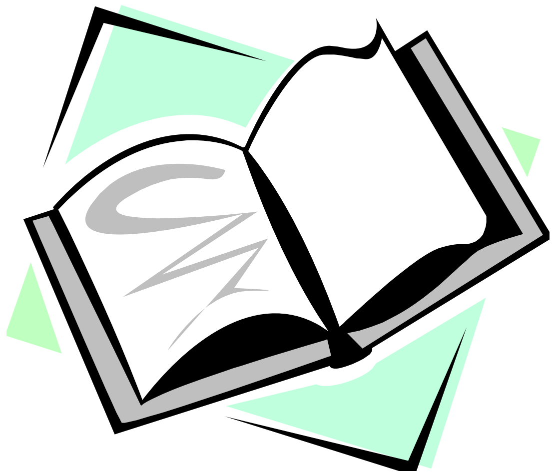
Differences with Other Integrity Models

- Other exceptions to formal integrity rules
  - low integrity objects can be upgraded to high by a high integrity subject
  - low integrity subjects can access high protected objects via exceptions

Readings for This Lecture

- Boebert & Jain: A Practical Alternative to Hierarchical Integrity Policies

- Li et al: Usable Mandatory Integrity Protection



Coming Attractions …

- IFEDAC & Windows Integrity Protection