Security Mechanisms and Policies

Andreas Zeller/Stephan Neuhaus
Lehrstuhl für Softwaretechnik
Universität des Saarlandes, Saarbrücken

SS 2004/CVS Revision: 1.4

The Menu

- Mechanism and Policy
- setuid
- How to Regulate Access
- The K.I.S.S. Principle

Subjects and Objects

Most access control is concerned with (active) *subjects* getting access to (passive) *objects*.

Process No. 123 opens `/etc/passwd`.

A web server, acting on behalf of user `zeller`, changes a row in a database.

An ATM, acting on behalf of you, reduces your account balance by some amount. . . and hopefully hands out some cash in the process

Mechanism and Policy (1)

Mechanism tells us *how* access control is enforced.

Policy tells us *which subjects* get access to *which objects*.

The Unix concept of file access permissions is (for the most part) a *mechanism*: it merely provides the method how to decide whether a process gets access to a file.

The system administrator and the users implement the corresponding *policy*, which is written down in /etc/passwd, /etc/group and the permission bits on the individual objects.

Practical guideline: If it is (or should be) in code, it's a mechanism. If it is (or should be) in a config file, it's a policy.

Mechanism and Policy (2)

It is important to separate mechanism and policy:

- Debug both separately
- Protect one from bugs in the other
- Change the policy without having to change the mechanism

It's important to separate the two *completely*:

Negative example: Unix. The Superuser always gets access.

This part of the policy is hardcoded into most Unix kernels, and becomes a de facto part of the mechanism.

That makes root omnipotent on a Unix system and that's why it is such an exposed account.

Access Control Matrix

View of access control mechanisms as matrix: row contains active objects (users, processes, etc.), columns contain passive objects (files etc.), matrix entry says what active object may do with passive object.

	passwd	httpd.conf	ls	Objects
root	r/w	r/w	r/w/x	
neuhaus	r	r	r/x	Capability
zeller	r	r/w	r/x	
cleve	r	r/w	r/x	
Subjects		ACL		

Full Access Control Matrix

A full access control matrix with n subjects and m objects has nm entries.

Typical Linux system has 100,000 files and directories in the root filesystem (not counting home directories, devices, mail spool etc.) and 30 entries in the password file ⇒ 3,000,000 entries, most of which are empty.

Every new user adds 100,000 entries.

Solution 1: Store entries by row: *capability*

Solution 2: Store entries by column: *access control list*

Solution 3: Store mostly columns, compute some rows: hodgepodge :-)

Properties of Capabilities

Capability systems issue a capability to a subject. When the subject wants to access an object, it presents the capability. This capability could be encrypted or otherwise protected.

- Can be given to other subjects (which can be good or bad)
- Can sometimes be computed on the fly (need not be stored)
- Can be issued minimally, i.e., just enough capabilities to do the job, but no more
- Revocation of capabilities and audit (who can do what) tricky
- Dissemination not easily controllable.

Crude example: passwords (no fine-grained control)

Properties of ACLs

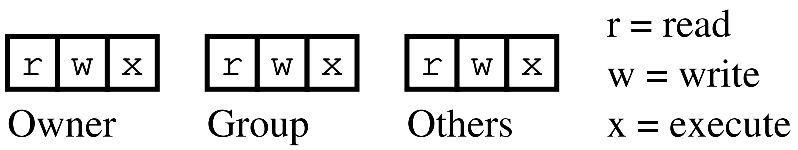
ACLs are dual to capabilities:

- ACLs cannot be transferred to other subjects (which can be good or bad)
- Must usually be stored with the object
- Creating minimal ACLs is difficult
- Revocation of access rights and audit easy
- Dissemination impossible ⇒ trivially controllable
- Privilege-granting privilege cannot be constrained

Crude example: Unix permission bits (see below)

The Unix File Access Model

Unix file objects have an owner and a goup. They also have nine bits associated with them (actually, there are twelve bits, we'll talk about the remaining three bits later).



Meaning of Bits

The first group of three bits tell what the *owner* of the object may do with it.

The second group of three bits tell what *group* members may do with it.

The third group of three bits tell what all *others* may do with it.

Within a group, bit 0 means "execute permission" (on directories: can change to this directory)

Bit 1 means "write permission" (on directories: can create new files, delete files)

Bit 2 means "read permission" (on directories: can read directory contents)

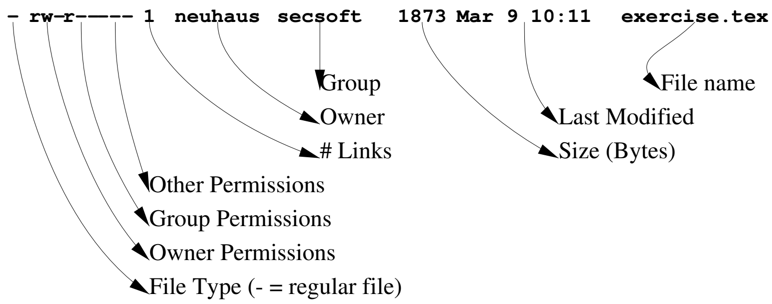
Access Control

When a file is opened (removed, renamed, executed, attributes changed), the process requesting the operation makes a *system call*.

The operating system examines the *effective user ID* (euid) and *effective group ID* (egid) of the process making the call.

It also looks at the file's (or directory's) *attributes* and thereby decides whether to perform the operation or not.

Example (1)



PID 123 has euid `zeller` and egid `secsoft` ⇒ access granted

Process 234 has euid `cleve` and egid `users` ⇒ access denied

Process 345 has euid neuhaus ⇒ access granted

Process 1 has euid `root` ⇒ access granted

Numerical Modes

Since there are groups of three bits each, we can express a mode in base 8:

Mode	Binary	Octal
`rw-r--r--`	110100100	644
`rw-------`	110000000	600
`rwxr-xr-x`	111101101	755
`rwxrwx---`	111111000	770
`--x--x--x`	001001001	111

Additional Bits (1): Sticky

Bit No. 10: Sticky bit. On regular files mostly without semantics; on directories: only owner can delete file, regardless of mode (used for temp directories)

Letter: `t` (if other x bit set) or T (if other x bit not set)

`/tmp` has mode 1777, `/tmp/strange` has mode 1770.

```
drwxrwxrwt  9 root    root   16384 Mar  9 13:01 /tmp
drwxrwx--T  2 neuhaus users   4096 Mar  9 13:39 /tmp/strange
```

Additional Bits (2): Setgid

Bit No. 11: Set group ID bit. On regular non-executable files, has mandatory file locking enabled (but don't count on it). On regular executable files, executes file with effective group id changed to group. On directories: Files created in that directory get group ID from directory, not from creating process.

Letter: s (if group x bit also set) or S (if group x bit not set)

```
-rwxr-sr-x  1 root    tty   9112 Jan 27  2002 /usr/bin/wall
drwxrwsr-x  8 zeller  www   4096 Jan 20 10:51 /home/www/edu/sopra
-rw---S---  1 neuhaus users    0 Mar  9 13:05 /tmp/lockme
```

Additional Bits (3): Setuid

Bit No. 12: Set user ID bit. On regular non-executable files without meaning. On regular executable files, this executes file with effective user id changed to owner of file, not owner of creating process. On directories semantics are unclear.

Letter: s (if user x bit also set) or S (if user x bit not set)

```
-rwsr-xr-x  1 root    root  23176 Apr  7  2002 /bin/su
-rwS------  1 neuhaus users     0 Mar  9 13:25 /tmp/testme
```

Sideshow: setuid Explained (1)

Unix is a multi-user OS. Different users have different numerical user IDs (called UIDs) and can be members of various *groups*, which also have different numerical group IDs (called GIDs).

One user has ultimate power over a machine. This is the *superuser* (aka *root*) which has UID 0.

When a new process is created, it inherits its UID and GID from the process that created it (the *parent process*).

The initial process (surprisingly called `init`) has UID 0.

In order to separate different users, a process running with UID 0 can set its UID with a system call (called `setuid()`). Once a program has given up its privileges, it can usually not revert to its old ones. This is usually a good thing!

Sideshow: setuid Explained (2)

When a program is executed, it is loaded (e.g., from disk), turned into a process, and started.

Some processes need more privileges than the parent process has.

For example, a process that changes a user's password needs superuser privileges because only the superuser can write the password file.

Some programs are therefore flagged with the additional information that any processes created from them need to have their UID changed to the owner of the program, and not inherited from the parent process.

This privilege elevation (and sometimes demotion) is handled by the kernel.

Mapping Unix to Access Matrix

The Unix permissions are (mostly) associated with objects

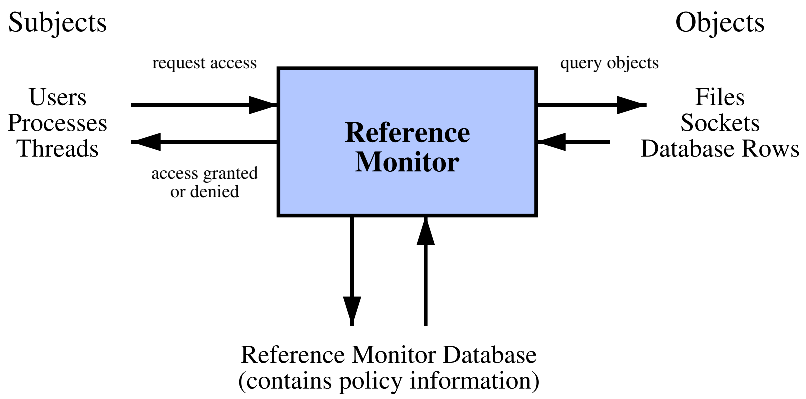
It is therefore (mostly) an ACL-like system.

The Access Control List then contains the owner explicitly, and the members of the file's group and everybody else implicitly.

Windows NT (and, by extension, XP and 2003) have real ACLs, where every object has a list of subjects and its permissions

Reference Monitors

The reference monitor is that piece of software that performs the access decision.



It implements the mechanism.

Properties of Reference Monitors

- Mediate *every* access
- Tamper-proof
- Simple enough to be analyzed comprehensively

Example: Unix File Manipulation

Remember: when a file is opened (removed, renamed, executed, attributes changed), the process requesting the operation makes a *system call*.

The system call goes through (a subsystem of) the kernel, who then decides whether to grant access or not.

- Mediates *every* access? Yes
- Tamper-proof? No, but protected
- Simple enough to be analyzed comprehensively? Not really

Security Policies

Definition: "The *security policy* of a system is a statement of the restrictions on access to objects and/or information transfer that a reference monitor is intended to enforce."

More generally: "The *security policy* of a system is any formal statement of that system's confidentiality, authenticity, and integrity (CIA) requirements."

Example: The Unix password and group files are part of a Unix system's security policy, because they determine

- when a user has successfully authenticated itself (A);
- which files the user can access (C)

Bell-LaPadula

Was designed in 1973 to codify existing military practices.

The system assigns a fixed numerical security level to each subject and object.

Needs a reference monitor that enforces two properties:

- The Simple Security Property: a subject may read only objects that are at its own security level, or lower ("no read up");
- The *-Property ("Star Property"): a subject may write only objects that are at its own security level, or higher ("no write down")

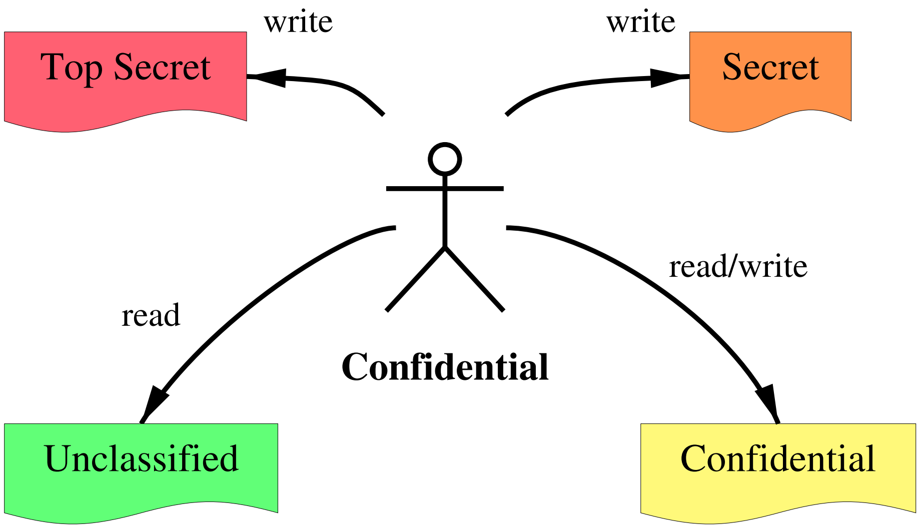
Origins of Bell-LaPadula

Written to address the "confinement problem", i.e., the unwanted dissemination of information by Trojan Horse programs.

Threat model: Trojan Horse program reads data on a multiuser mainframe and writes it to a location where an outsider can read it.

That's too oldfashioned? OK: a Word macro virus steals your sensitive data and uses Outlook express to send it over the Internet.

Example



Practical Problems with Bell-LaPadula

Information tends to flow upward to the highest security level.

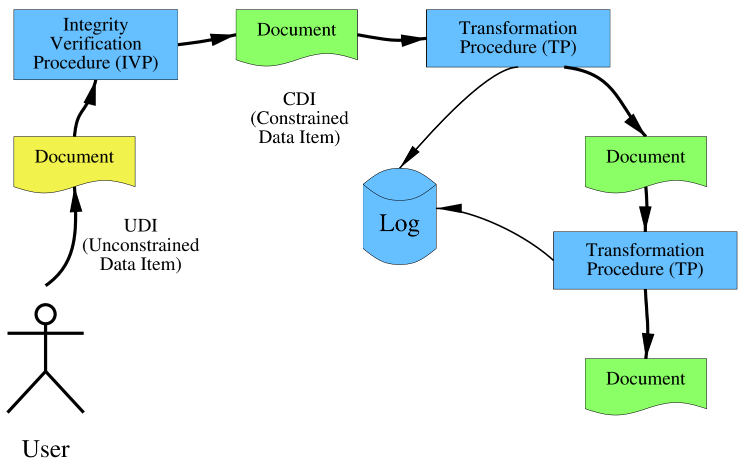
- Can't move down again because of *-property
- Once a document is accessed at level n, it can't be returned to level $m < n$.
- Users tend to have multiple copies at different levels.
- Also bad for integrity: Writes can't be verified.

Problems with Email and related services:

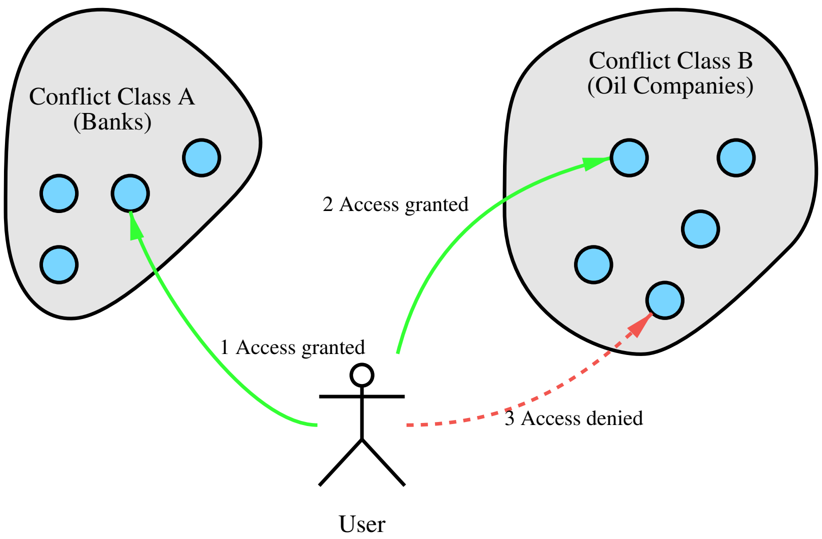
- A user level m isn't made aware of email messages at level $n > m$.
- A user at level n can see messages at level $m < n$, but can't reply to them

Clark-Wilson Model

How would a model look that is more concerned with *integrity* instead of confidentiality?



Chinese Wall Model



Chinese Wall Explained

Object groups are partitioned into distinct *conflict classes*.

“Partitioned” means that an object group belongs to exactly one conflict class.

A subject can get access to any object in any group, provided that it does not already have access to objects in another object group in the same conflict class.

Access to objects within an object group is unrestricted.

Under these conditions, a conflict of interests cannot occur.

Practical problems abound; see exercises.

Problems with Policy Models

It turns out that all these above policy models are equivalent to Bell-LaPadula. (We won’t prove this.)

“These basic models were intended to be used as general-purpose models and policies, applicable to all situtaions in which they were appropriate. Like other flexible objects such as rubber screwdrivers and foam rubber cricket bats, they give up some utility and practicality in exchange for their flexibility, and in practice tend to be extremely difficult to work with.” — Peter Gutmann

Solution: Apply policy models only to small and specific parts of the entire system. *Don't look for a silver bullet!*

Implementation of Mechanisms

Implementation is very specific; therefore we will analyze a particluar implementation.

Implementation is that of *cryptlib*, a cryptography toolkit written by Peter Gutmann.

Design goals:

- Must run on many architectures (VAX, IBM mainframes, embedded systems);
- Must support crypto hardware
- Must support many good crypto algorithms under a single unified interface
- Must present a *secure interface* to the user, one that is impossible to use in an insecure manner.

Why cryptlib?

- It's there to be analyzed (free for academic use)
- Was designed with explicit goals in mind (not a hodgepodge of useful routines with questionable code quality like OpenSSL)
- It's secure (has had zero(!) security problems since its inception in 1992)
- It's well documented (comes with 300+ page user manual and tutorial; design and implementation are described in Gutmann's 300+ page Ph.D. thesis)

Cryptlib Mechanism Architecture

How to verify the implementation? The entire implementation (C source and header files) is about 215,000 lines of code...

Answer: by decomposing the system such that the components have no direct interaction or interact only with similar components.

This is in contrast to a typical object-oriented decomposition where objects have interaction with *all manner* of other objects.

This architecture is more like a *virtually distributed* system: The objects are not really distributed, but they could be.

Access to objects is mediated through a trusted *security kernel* that forms (part of) the *trusted computing base*

Brief Sideshow: TCB

The *Trusted Computing Base* (TCB) of a system is that part of it that must be able to make certain security guarantees in order for the entire system to be secure.

Like an axiom in mathematics.

Typical components of of the TCB: Hardware (processor, RAM etc.)

Operating System (ouch!)

Is Linux verifiable? Windows? Certainly not!

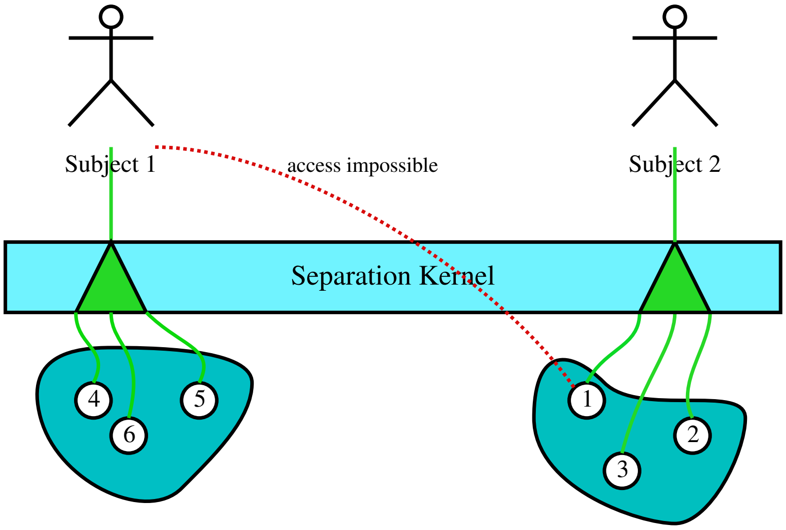
Security Requirements

- The kernel must enforce the security of the system as a whole without requiring its components to cooperate towards that end.
- No input or output to or from any object can interfere with any input ot output to or from any other object
- Policy: A subject can only access objects that it owns. (Any active entity can be a subject in this system, not just users.)

⇒ No concept of object sharing

⇒ No concept of security levels

Separation Kernel



Access Mechanism

With a separation kernel, the access mechanism becomes almost trivial:

> Any object is labeled with the id of the (unique) subject that owns it. If any subject wants to access any object, the subject's ID is compared to the object's owner ID. If they are equal, the subject is granted access, otherwise access is denied.

All accesses to objects by subjects is also mediated by the kernel ⇒ easy to enforce the policy.

An easy formal proof is also possible.

"[A] lot of security problems just vanish and others are considerably simplified" (John Rushby)

Additional Policies

- No ability to run user code. Simplifies implementation and verification.
- Single-level object security: No information sharing between subjects; no multi-level security. Simplifies implementation and verification.
- Multilevel object attribute and object usage security: uses ACLs to react to messages that modify an object's state.
- Serialization of operations. Lets kernel mandate when messages can be passed to objects; no need to check in each object's implementation.
- Object usage controls: control purpose, number of uses etc., so that e.g. a signing key can be used to create exactly one signature.

The K.I.S.S. Principle

In other words:

Keep it Simple, Stupid!

Simple systems are easier to analyze and understand than complicated systems.

Complicated systems are more prone to errors: "I have found that I can trust only code that is easy to understand; the bugs are almost always in places where I try to be clever." (Wietse Venema, author of postfix)

> "There are two ways to design a system. One is to make it so simple that there are obviously no deficiencies. The other is to make it so complex that there are no obvious deficiencies"
> —C.A.R. Hoare

Why Enforce Policies?

Why enforce policies in the kernel? Why not write the individual subjects so that the policies are never violated?

Because then all the policy decisions are all over the place and cannot be easily verified.

Also, a malicious subject could subvert the controls.

Collaborative Security Code

Can secure code be written in the style of typical open-source projects?

“Many eyes make all bugs shallow” (Eric S. Raymond)

“Hmm, I could [review other’s code] but I doubt I’d be able to be anywhere near as thorough as on my own code. I also know what my code is supposed to do and what to expect, whereas [other person]’s code will have his own security design and expectations, so I guess we could end up checking a lot of stuff that doesn’t really need to be checked.” (Peter Gutmann)

Summary

- Access Control
- Access Control Lists/Capabilities
- Bell-LaPadula
- Chinese Wall
- The cryptlib Separation Kernel
- Trusted Computing Base
- The K.I.S.S. Principle

Resources

- Peter Gutmann, *Cryptographic Security Architecture*, Springer
- Peter Gutmann, *cryptlib Encryption Toolkit*, `http://www.cs.auckland.ac.nz/~pgut001/cryptlib/`