a

Security mechanisms in IX

IX, a species of Unix, was made in the late 1980s by Doug McIlroy and Jim Reeds. It provided

- Mandatory multilevel security, enforced even on superuser
- Dynamic security labels for both secrecy and integrity
- Confined exercise of privilege
- Safe networking with untrusted systems
- Structured administration of privilege

IX is not an ‘‘orange-book’’ system. It has

- wider covert channels
- no access control lists

But, it does cater for

- evolution of the TCB
- highly systematic expression of security policy

The easy part: dynamic labels

Every process, file or channel has a security *label,* chosen from a (slightly augmented) lattice.

Normally data must flow *up.* Flow is allowed from x to y only if their labels satisfy

$$L(x) \leq L(y)$$

The lattice is augmented by two special labels:

yes
All flows are allowed. `/dev/null` is labeled **yes**.

no No flow is allowed. External devices (communication ports, disk drives, printers, etc) have label **no** when not in use.

No subject/object mumbo-jumbo

Though subjects may exist outside the system, the IX model is only concerned with data flow and labels.

Dynamic labels

Labels are compared at every system call that entails data flow.

For flow from x to y:

If $L(x) \le L(y)$ the transfer proceeds.

Otherwise, if possible, the destination label is changed to the least $L'(y)$ for which

$$L(x) \le L'(y)$$
$$L(y) \le L'(y)$$

Otherwise, the transfer fails.

The destination label cannot be changed if

The new label would violate a specified *ceiling*: $L'(y) \not\le C(y)$.

The label is unchangeable, either by a user's choice, or by system fiat.

Degrees of fixity of labels:

floating	
fixed	
rigid	changeable only with privilege (`/dev/tty`)
constant	unchangeable (`/dev/null`)

Form of labels

Labels are 64 bytes, comprising

- Lattice node, a 480-bit vector.
- **yes/no** indicator.
- Bit vector of privileges.
- Bit vector of audit categories.
- Fixity indicator.

Ceilings

Ceilings are represented as labels.

Process ceiling may be dropped at will, but raised only with privilege.

File system ceiling is imposed at mount time.

Privilege part masks file privileges.

Encoding labels for various purposes

Military-style classification:

unclassified	000
confidential	001
secret	011
top secret	111

Simple compartments: use one bit per compartment.

To go beyond 480 compartments:

- Uncombinable compartments: use n-bit codes of weight $n/2$

  $\binom{n}{n/2} \approx (2\pi/n)^{\frac{1}{2}} \cdot 2^n$ distinct compartments

- r-way combinable compartments: use r-cover-free families.

Different encodings may be applied in different parts of the label space.

Initial process label

At `exec`, a process gets the label of its executable file.

The label rises if necessary to cover arguments passed by `exec`.

Shells always pass some arguments. A simple utility

> `drop` *program*

invokes *program* with no arguments, so the process label can drop to that of the program file.

Examples of reading and writing.

`Creat` writes in a directory.

`Chmod` writes in an inode.

`Stat` reads an inode.

`Lseek` writes and reads a seek pointer.

> Exception: the return value of `lseek` relative to beginning of file has no information value.

Shared seek pointers

Data can flow through a shared seek pointer, so seek pointers have labels of their own.

> *Why not give the seek pointer the same label as a file?*

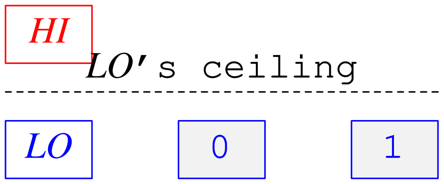
IX also reinstates two ancient system calls `seek` and `tell,` which separate the write and read functions.

Covert channels

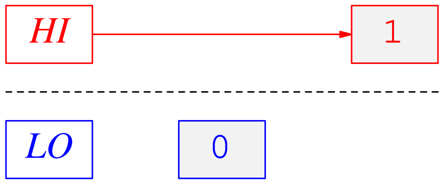
Communication can occur without explicitly passing bits, e.g. through error returns.

A classic trick to pass one bit from process *HI* to process *LO*:

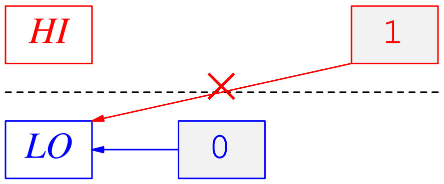
1. *LO* creates files `0 and 1.`



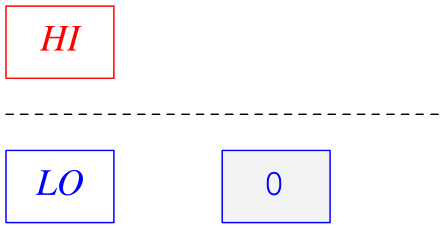
2. *HI* writes in one of them



3. *LO* learns which is unreadable



4. *LO* deletes unreadable file



A wider covert channel

This scheme communicates up to lg v bits per exec, where v is the size of a preset vocabulary.

Directory `vocab`, labeled low, contains a program for each word of the vocabulary:

file	shell script
`attack`	`echo attack >>˜spy/message`
`at`	`echo at >>˜spy/message`
`dawn`	`echo dawn >>˜spy/message`
`...`	`...`

A mole can smuggle out a message thus:

```
for word in  attack at dawn
do
    drop vocab/$word
done
```

Maybe drop-on-exec isn’t such a great idea.

Tolerance for covert channels.

The example is typical. Most covert channels in IX involve creation and deletion of files and/or processes for each few bits transferred.

These relatively slow operations hold the bandwidth down.

Extensive use should reveal itself in anomalous audit profiles.

A patient Trojan horse unwittingly used by an authorized party might smuggle out small secrets via such channels.

But wholesale data larceny is likely to be detected.

Boebert's LOCK system closed these channels by putting each label in a parallel universe, with copy-on-label-change—a more radical tradeoff between security and Unix functionality.

IX rule of thumb: 100bps channels are tolerable.
Auditing may drive the bandwith lower.

Integrity labels

Biba pointed out a duality between integrity and secrecy:

> Reading of high-secrecy data is strongly controlled.
>
> Writing of high-integrity data is strongly controlled.
>
> An integrity lattice is essentially an upside-down secrecy lattice.

To use the label lattice for both integrity and secrecy, we give unwashed processes labels below high-secrecy and above high-integrity.

IX sessions begin at the *floor,* a label at the top of a sublattice dedicated to integrity.

label	hex value
top	`ffff ffff ...`
floor	`ffff 0000 ...`
bottom	`0000 0000 ...`

Files in the trusted computing base (TCB) are given labels below the floor. They may be marked privileged to prevent writing even by a superuser with a low label.

Efficiency

Label checks impose a significant overhead on system calls.

Read and write checks are complex. They involve comparisons of, and sometimes changes to,

- Process label
- File label
- Seek pointer label
- Process ceiling
- File system ceiling

A full label check at every read and write would impose a huge cost.

The overhead on read and write is reduced by caching checks, so that normally only a single “safe-to-read” or “safe-to-write” bit must be checked.

Propagating label changes

When a full label check for `read` (or `write`) succeeds, the file descriptor is marked safe for reading (or writing).

When a process's label rises, all its file descriptors are marked unsafe to write.

When a file's label rises, associated file descriptors in all processes are marked unsafe for reading and writing.

When a pipe's label rises, file descriptors at both ends are marked unsafe.

A data structure of pointers allows for quick propagation of unsafe marks.

> Cost: typically more than 100KB of physical memory for pointers and file descriptors, which now can't be paged out.
>
> Benefit: overhead for reading and writing is too small to measure.

The hard part: privilege

Some actions require the label rules to be broken:

- Downgrade (declassify) a file.
- Start a session: identify the rights of a supplicant at a port; set the port label and process ceiling accordingly.
- Replace TCB files.
- Mount a filesystem.
- Multiplex data of different labels over a channel to a trusted peer.
- Check a filesystem (`fsck`).
- Administer privilege.

The privilege mechanism.

Consider `nochk`, the privilege to ignore label checks.

A process p has a boolean "capability" Cap(p) and a boolean "license" Lic(p) for `nochk`.

The capability signifies the actual right to ignore label checks.

The license, but not the capability, is inherited across `exec`.

A file f also has a capability Cap(f) and license Lic(f).

Changing a file clears its privileges.

Determinining capability

Process p executing file f gets `nochk` capability only if the file has the capability and the process is licensed for it. To a first approximation:

$$\mathrm{Cap}(p) = \mathrm{Cap}(f) \cap \mathrm{Lic}(p)$$

A file license is like setuid. It applies to a process that executes the file.

Privilege is exercised by trusted tools in trusted hands.

To prevent the import of unwanted privilege from a mounted filesystem FS, ceilings $\mathrm{Cap}(FS)$ and $\mathrm{Lic}(FS)$ may imposed.

In gory detail:

$$\mathrm{Cap}(p) = \mathrm{Cap}(f) \cap \mathrm{Cap}(FS) \cap (\mathrm{Lic}(p) \cup \mathrm{Lic}(f) \cap \mathrm{Lic}(FS) \cap \mathrm{Lic}^0)$$

(Lic^0 is a compile-time parameter that enables self-licensing.)

Privilege categories.

Categories tell what rules can be broken:

- `Nochk`. Ignore label comparisons. To help avoid mistakes, a nochk process must declare what file descriptors it needs the privilege for.
- `Extern`. Change labels away from `no`, and change labels with “rigid” fixity. Used to set the labels on external ports.
- `Setpriv`. Change file privileges.
- `Setlic`. Set the process license, label and ceiling.
- `Log`. Control mandatory auditing.
- `Uarea`. Allow changes to process-readable system data, normally set by the superuser, e.g. userid and login name. Items controlled by ordinary users, such as `umask`, have their own labels. Perhaps the superuser items should, too.

Privileges have one-letter abbreviations: `n x p l g u`

Private paths

A “private path” mechanism can be used to assure the bona fides behind a file descriptor and prevent other processes from interfering with transactions on the path.

Some uses of private paths

- to collect passwords at login
- to assure faithful copies when installing TCB programs

The operation `pex` asserts privacy at one end of a stream.
The stream cannot be used until pex has been accepted at the other.

For a file, acceptance is automatic.

For a pipe, acceptance must be overt.

`Pex` exchanges indicia of trust (i.e. privilege), which the processes may use in deciding whether to continue.

An application of `pex`:

```
pcopy source destination
```

(paranoid copy) prevents mischief from interfering programs. It reproduces as much inode info as possible, including label and privilege.

Stream identifier

A stream, such as a terminal port, may carry transactions for many processes that need to know security-related information such as who's at the other end, and is the remote computer trusted.

Such information, encoded in a string called a *stream identifier,* may be attached to a stream.

To prevent forging, the stream identifer can be set only with privilege.

System administration

Major ingredients

Privilege server, to vet and grant requests to exercise privilege.

`Priv`, to run a program with privilege; a wrapper for the priv server.

`Nosh`, a no-surprise shell.

Example of running a privileged program

`priv downgrade` *bits file*

asks the privilege server to run a program that will downgrade certain bits of a file's label.

Before invoking an action, the priv server will

- Authenticate the user
- Check the right to do the action
- Get confirmation that this was the desired action.

Priv server

The priv server is the keeper of security policy.
It hands out exact privileges for specified tasks.

A policy file (`/etc/privs`) contains

- an authorization tree
- a set of actions

Nodes in the authorization tree grant *rights,* subject to an *access predicate*

Actions tell how to carry out particular requests from clients found to have specified rights.

Rights

A right is a local identifier, meaningful only to the priv server.
It may be restricted to some part of a lattice:

`LAB`	the lattice of labels
`PRIV`	the lattice of subsets of privileges
`RE`	the lattice of regular languages ordered by inclusion

Typical rights declarations

```
DEFINE sublattice(LAB), privinstall;
DEFINE filename(RE);
```

Rights are monotone in the tree: a child node has a subset of the rights of its parent. The tree's root has all rights.

Rights are granted at tree nodes designated by path names:

```
RIGHTS /admin              priv(upnxl), sublattice(ffff...)
RIGHTS /admin/security     priv(p), sublattice(ffff...)
RIGHTS /admin/operations   priv(xn)
```

Access predicates

An access statement expresses a conjunction of predicates that contain regular expressions:

`ID(`*names*`)`	login name
`PW(`*names*`)`	check password
`SRC(`*streamid*`)`	check stream identifier on std input

Example

```
ACCESS /admin   ID(wbc) PW(wbc);
ACCESS /admin   SRC(console);
```

To get `/admin` authorization, you must either have login name `wbc` and be able to present `wbc`'s password, or be at a (secured) console.

Action rules

Given a request, which looks like a shell command, the priv server searches for tree nodes whose access predicates are satisfied.

It then matches the request against action rules of the form

REQUEST(*words*) NEEDS *rights-list* DOES *action-list*;

The request can be acted on if the needed rights have been granted by some tree node.

The request is typically a prefix of a shell command.
Words of the command can be referred to by shell-parameter syntax.

Example. The privilege request

```
priv downgrade 0100... myfile
```

matches

```
REQUEST(downgrade)      NEEDS sublattice($1)
    DOES PRIV(nx) EXEC(/bin/downgrade $1 $2);
```

Other legitimate actions: start a daemon, set the process ceiling, edit some subtree of the priv server.

To allow requests to come through untrusted hands such as `/bin/sh`, the server echoes the request (under cover of pex) for the client to confirm.

The action is then performed hygienically, with an empty environment and a "black hole" current directory.

No-surprise shell

`Nosh` lacks a horde of exploitable features.

Path search; every non-builtin command name must contain `/`.
Variables, environment, functions, aliases
User profile
Redirection of files other than standard output.
Compound commands: pipelines, `if`, `for`, etc.
Background commands (`&`)
Manipulation of signals
Argument generation (`*  ` `...` ``)
History, command editing
Most builtins, arithmetic, test

The `nosh` program file has all capabilities, but is not self-licensed.

By default `nosh` does not pass licenses to child processes. A builtin function `lmask` does that.

```
lmask x setlab ffff0000... 911report
```

reclassifies a file to the floor, provided `nosh` already has license `x`.

A running system will normally count on the priv server, not `nosh`, to license commands.

However, `nosh` would be licensed in the boot script and in emergency single-user mode at the console.

Paranoid updates

A two-key method for installing a privileged program P by joint actions of system administrator and security administrator.

1 Sysadmin compiles or downloads new program P'.

2 Sysadmin protects P' by giving it a dummy license.

3. Sysadmin vets* P'.

4. Secadmin adds another dummy license that sysadmin cannot.

5. Secadmin vets P'.

6. Sysadmin removes privilege from P and copies P' (including the dummy licenses) to P using `pcopy`.

7. Secadmin compares P and P'.

8. Secadmin sets the correct privileges on P.

9. Either removes the dummy licenses from P'.

* Vetting may be done by acceptance testing, comparing, decompiling, scanning for viruses, checksumming, etc.

Final remarks

IX is modest. It comprises about 6000 lines of new or changed code in kernel and other programs. Half of that is in the priv server.

Major lacks

- a symbolic way to refer to labels
- audit-analysis tools

Triumph: IX security nabbed a virus, without any overt virus checks.

Learning experience: tar-baby directories.

IX demonstrates

- Dynamic labels, which partially insulate Unix semantics from security-labeling concerns, can be implemented efficiently.
- General labels; no built-in distinction between level and compartment.
- Safe networking with trusted and untrusted actors, (`pex`, **no**, (*FS*)).
- Privilege defined by what rules are broken, not what roles are played.
- Priv server to support structured security policy, including controlled delegation of privilege.

IX is designed to protect against automated theft or vandalism by unauthorized users, but not against laborious leaking of data by dishonest programs in the hands of authorized users.

We believe the data-flow model better describes office- and network-security policies than does the subject/object model.