Multilevel Security in the UNIX Tradition

M. D. McILROY AND J. A. REEDS

ABSTRACT

The original UNIX® system was designed to be small and intelligible, achieving power by generality rather than by a profusion of features. In this spirit we have designed and implemented IX, a multilevel-secure variant of the Bell Labs research system. IX aims at sound, practical security, suitable for private- and public-sector uses other than critical national-security applications. The major security features are: private paths for safe cooperation among privileged processes, structured management of privilege, and security labels to classify information for purposes of privacy and integrity. The labels of files and processes are checked at every system call that involves data flow and are adjusted dynamically to assure that labels on outputs reflect labels on inputs.

1. INTRODUCTION

We have built *IX,* an experimental 'multilevel secure' variant of the UNIX operating system. IX supports document classification with *mandatory access control*; classified input must yield classified output. Its security model differs from that of Bell and LaPadula, which is espoused in the National Computer Security Center's 'Orange Book.'[1, 2] This paper is an overview; details are given elsewhere.[3, 4, 5]

IX preserves data classification. Every file and every process has a *label,* which tells its classification. Users are allowed to see only information they are cleared for. Data transfers may only happen in a direction of increasing labels. Labels of processes or files may adjust automatically during computation to guarantee that outputs are classified at least as high as the inputs from which they derive. Labels are discussed more fully in §2.

IX clips the wings of the superuser. Activities, such as declassification, that deviate from the usual labeling rules can be accomplished only with the exercise of *privilege.* A trusted user may be endowed with one or more privileges, which may be exercised only through trusted programs that have been certified for those privileges. In normal usage each use of privilege is vetted by a *privilege server,* which confirms the client's authority and hands out exactly the privileges needed for the operation at hand. The basic privilege mechanism is described in §3; its use is further explained in §6.

IX provides private communication paths and methods for mutual confirmation between privileged processes (§4).

IX safeguards outside communications. External media such as tape cartridges or communication ports can be opened and labeled only by trusted code. That code has the duty to authenticate the clearance of the destination. After trusted code has set the label, an external medium can be used like any other file. In particular network connections, once established, are as easy to use as in ordinary UNIX systems. We routinely cross-mount the file systems of IX machines and ordinary UNIX machines for the exchange of unclassified data.

There is little experience with multilevel systems in nonmilitary applications. We expected that we would learn by trying to make one, and that we would learn more by trying a model that was not literally Orange Bookish. Wishing to preserve as much of the flexibility and spontaneity of UNIX as possible, we have taken less draconian measures against covert channels than the Orange Book suggests (§2.2). Thus, IX will protect information from automated theft by unauthorized users and from accidental disclosure, but will not perfectly protect it from being leaked laboriously by dishonest programs run on behalf of authorized people.

We wished particularly to preserve the simplicity of programming in the large with shell scripts and pipelines. In support of this goal, dynamic labels eliminate some of the need to foresee just what labels a run might produce. A potential benefit is more accurate labeling, for output files can be labeled exactly, not merely with a convenient umbrella label.

In short, the IX model, unlike Bell-LaPadula, was intended to make security calculations for users rather than against them.

1.1. Data-flow versus subject/object models

Modern computer systems, where files may be fronts for server processes and processes may act on behalf of no person, accord poorly with the Bell-LaPadula subject/object model. Roughly speaking, that model describes the computer as a filing cabinet plus a collection of isolated *subjects* who visit it to consult or deposit *objects.* The subjects, usually processes understood as proxies for real people, are branded with security clearances. The objects are usually files. A *reference monitor* guarding the cabinet interdicts access to secret files by uncleared people or deposition of public files by cleared people. At the beginning of each computer session, or 'day at the office,' a person must select a legitimate clearance and stick to it. Ongoing activities involving data at different classification levels are constrained to run at the highest of those levels, much as if a lunch order from a classified meeting had to be classified. Interprocess communication complicates matters. Transactions then happen without going through the filing cabinet. It becomes necessary to invent subjects unconnected with persons and to identify some subjects also as objects.

IX caters for more realistic 'office protocols' than subject/object models do. It simply recognizes places between which data occasionally flows. Data flowing out from a place that has memory is contaminated by data that has flowed in; hence data labels must be tracked from place to place. Nothing, however, prevents a succession of actions from happening at different levels. Lunch orders can leave classified meetings, albeit not utterly freely, because the computer is charged with assuring that the lunch order not be written on the back of a secret report. What ultimately counts is that data leaving the computer should reach only agents who are eligible to receive it. The agents (subjects) do not appear in the model at all; but their limitations appear as constraints on the labels of data flowing to output ports.

In Bell-LaPadula systems the label of a file must remain constant while the file is in use, so labels need be checked only when files are opened. In IX, by contrast, where labels of files and processes change underfoot, labels must be checked on every data transfer. Continuous label checking posed a challenge: to check labels without incurring unacceptable overhead. It also provided reassurance; nothing depends on the fiction that labels never change. No special system mechanisms are needed to prevent untoward side effects arising from a change in the label of a file or of a terminal session.

In retrospect, continuous label checking was not hard to do. Privilege, for which the literature offers no models, was a more recalcitrant matter. We have found ourselves ever more concerned with confining the use of privilege, establishing mutual trust among cooperating privileged processes, and guaranteeing the integrity of their communications. These concerns were addressed by the notion of private communication paths (§4) and a structured privilege server (§6.1).

2. LABELS

Every file, device, and pipe has a classification label, as does every process. For technical reasons, every seek pointer, which gives the current location in an open file, also has a label (§7.2). Furthermore, every process and every file system has a *ceiling,* a label below which all transactions must stay. File system ceilings help in managing remote file systems and exportable media. Process ceilings are a kind of insurance. They partly fulfill the function of Bell-LaPadula subject labels—preventing processes from getting into overly sensitive places, from which they could leak data by covert channels. They also prevent the injection of noise by writing into high places, and inadvertent excursions to high level that result in overclassified outputs.

A label is an element of a mathematical lattice (§2.6) or one of two special symbols, **yes** and **no**. Label **yes** is intended for files that may always be read or written, notably `/dev/null`. A file labeled **yes** is perfectly amnesiac; what comes out is unrelated to what goes in. Label **no** is intended for files that cannot be read or written without the intervention of privilege. Every external device file (terminal, communication link, disk, etc) is labeled **no** when not in use.

The label of an entity x (process, file, or seek pointer) is denoted $L(x)$. The ceiling of an entity (process or file system) is denoted $C(x)$. Labels may vary with time. We write $L(x,t)$ or $C(x,t)$ when the time t matters. Labels may be compared. The inequality $L(x) \le L(y)$ holds when either x or y is labeled **yes** or when $L(y)$ dominates $L(x)$ in the lattice.

2.1. Data-flow policy

Data flow results from system calls. In general an unprivileged process can only cause 'upward' data flow below all pertinent ceilings. When data flows from source x to destination y, it is required that $L(x) \le L(y)$. Moreover, if the causative process is p, then $L(y) \le C(p)$. If x (or y) belongs to file system z, then $L(x) \le C(z)$ (or $L(y) \le C(z)$).

Data also flows in time. In general the label of any memory x must satisfy $L(x,t_1) \le L(x,t_2)$ for $t_1 \le t_2$. The ceiling of an unprivileged process must decrease monotonically with time: $C(p,t_2) \le C(p,t_1)$ for $t_1 \le t_2$. The ceiling of a mounted file system cannot change.

The label of a memory may be reset when it is reinitialized, that is, when its entire contents are replaced. One reinitialization action is argumentless execution of a file, which replaces process memory (§2.3). Another is absolute file seek, which replaces a seek pointer. File truncation, however, is not deemed to be a reinitialization, because some settable properties of a file (owner and permissions) persist across the operation.

We understand data flow to be a direct transfer of bits. Bits originating at a source are received unchanged at the destination. Other mechanisms of communicating information are deemed to be covert channels (§2.2).

Reads and writes constitute data flow, as does creation of file names in a directory. A file seek may participate in data flow because the seek pointer can be read. The customary permission bits of a file and the so-called modification date, which can be set arbitrarily, may participate in data flow. On the other hand, the inode number (serial position in the file system) and the file change date, neither of which can be set directly, do not participate in data flow. Directly readable kernel data, such as login names and userids, participate in data flow; other kernel data, such as the table of open files do not. Although error returns from system calls do not constitute data flow, process exit status does; status is censored to a one-bit success/fail indication unless the flow is upward.

2.2. Covert channels

Methods of communicating information that are not recognized as explicit data flow are called *covert channels.* Covert channels typically involve the modulation and observation of usage patterns, for example of file openings or data accessibility.

If a spy could get a process with a high enough label, the spy could read nuggets of information and smuggle them out via covert channels. For example, the fact of a file's being classified can be used to communicate information: a high process conditionally writes in a file and a low process detects whether the write happened by attempting to read it. The read fails if the write occurred. One bit of information has been communicated at the cost of a file creation, a write, a read, and a file deletion to wipe the slate clean.

Process ceilings guard against intrusion by preventing uncleared users from obtaining access to high places. A cleared mole or a Trojan horse, however, can leak via covert channels. Trojan horses may be countered in several ways. The ceiling of highly cleared users can be kept low during ordinary work, so that a process label cannot rise to unintentional heights. Integrity labels (§2.7) can be used to protect highly cleared users against executing unapproved software. Static auditing can detect mutations in programs. Dynamic auditing can reveal the exotic behavior of programs exploiting covert channels, most of which involve an unusual ratio of file or process creation to other activities.

We have determined the typical bandwidth of covert channels, and have closed channels of significant bandwidth whether or not they involve direct data flow. Some covert channels have been closed for reasons other than bandwidth. For example, neither deleting from nor searching in a directory entails overt data flow. Nevertheless labels are checked on search to frustrate prowlers, and on deletion to prevent meddling above a process ceiling.

2.3. Anti-inflationary measures and their dangers

To help keep labels as low as possible, IX has a 'drop-on-exec' feature. An 'empty' process gets the bottom label, which will later rise as usual to cover the labels of code that the process executes and data that it reads. A process is deemed empty if it has no arguments and has no open files beyond the standard four (input, output, error, and control). With drop-on-exec, a user in a high session can print a low document without gratuitous overclassification, thus:

```
runlow lp <low.doc
```

Runlow suppresses the hidden arguments—command name and environment variables—that UNIX shells normally pass to invoked commands. If the arguments were passed, they would contaminate *lp* and keep its label at the same level as the shell.

Drop-on-exec entails covert channels, most notably through the identity of open files. It is not clear that the feature is worth the price (§7.4), for it can be used only in circumstances where one process can set up another without passing arguments. In particular, drop-on-exec is incompatible with the UNIX convention of environment variables.

At one time, IX created each new file with a bottom label, in an attempt to avoid label inflation when copying low files. This convention, however, allowed data flow through the file access mode bits, and was nearly useless because a low file created by a high process would necessarily be hidden in a high directory. The convention has been abolished in the name of purity.

2.4. Fixed labels

When the name of a new file is written in a directory, the label of the directory may have to rise to cover that of the creating process. Should this happen unexpectedly to a low directory, the whole directory can go out of sight for applications that had been using it. The directory becomes a 'tar-baby':[6] any process that searches or reads it is forced to a high label. If the process makes files in other directories, it may tar those directories, too. To forestall such behavior, the owner of a directory (or file or process, for that matter) may mark its label 'frozen.' As long as a label is so marked, it cannot be changed and transactions that would have caused a change are aborted.

The label policy implies that the labels of some files cannot change freely. In particular the labels of external media, such as terminals, tapes, and communication links, must not rise arbitrarily, lest output exceed the clearance of the destination, which is not under control of the local system. Such labels are irrevocably marked 'rigid,' meaning they can be changed only with privilege.

2.5. Implementation of labels

The label check performed at a system call depends on whether the call involves a file name or a file descriptor (the handle by which a process accesses an open file), whether the call is write-like or read-like, and so on. Most checks fall into a few classes that can be dispatched by a table. The classes differ mainly in what dominance relations may be affected. For example, the *read* system call must respect seven dominance relations among five labels: file label, process label, seek pointer label, process ceiling, and file system ceiling. The process and seek pointer labels may possibly be adjusted to preserve the relationships. The dominance relations and adjustabilities are recorded in tables that are interpreted by a single generalized label-check subroutine, which finds the minimum legal adjustment needed to satisfy the checks, or backs out gracefully if adjustment is impossible.

Continual rechecking of labels in this degree of generality is a time-consuming matter (§7.1). Fortunately labels don't change often, so we use a check-caching scheme to avoid unacceptably large overhead from comparisons. Every file descriptor is endowed with two 'safe bits,' one for safe-to-read and one for safe-to-write. A network of pointers, some of which exist in traditional UNIX implementations, and some of which are new, connects files, file descriptors, and processes. Whenever a file label rises, associated safe bits in all processes can be located quickly and turned off. Subsequent accesses via other file descriptors will find the safe bits cleared and cause the label relations to be checked in detail. The label relations will be reestablished if possible, and the safe bits turned back on. Similarly, when the label of a process rises, safe-to-write bits are turned off throughout the process.

Typically the safe-to-read bit in a file descriptor will be turned on by the first read after file opening and stay on thereafter. As long as the safe-to-read bit doesn't change, label checking amounts to a one-bit test in each read (or in each atomic data transfer of a long read that the system chooses to do in pieces). Thus dynamic label checking costs imperceptibly more in time than would static label checking at file open, which suffices in Bell-LaPadula systems. Space costs, however, are significant (§7.3).

2.6. Representation of labels

Labels in IX comprise lattice values, privilege bits (§3.1), two bits for fixity (modifiable, frozen, rigid, unmodifiable; §2.4), and an indicator for **yes** and **no**. Lattice values are represented as bit vectors; lattice domination is represented as set inclusion. No further structure is predefined. Notions such as security level, compartment, or Biba integrity (§2.7) are trivial to encode, however.

Labels account for about half of all the descriptive information kept permanently with each file. The labels of open files occupy appreciable space in main memory and take appreciable time to compare.

We considered, and for several reasons rejected, a scheme of indirect labels in which each file bears a pointer into a table of all existing label values.

1. It could be harder to recover from an inconsistent state induced by a crash or otherwise.
2. The corruption of one table entry would mislabel a bundle of files at once.
3. To fit with existing backup schemes, backup files would carry full labels anyway.
4. To realize the full benefits of indirect labels, there should be a separate coding table for each removable file system.
5. With dynamic labels, the encoding of labels would evolve differently on every system, even systems under common administration.

The first reason was the most compelling. With directly labeled files, inconsistencies between label and data are confined to individual files. Moreover, the inconsistencies are harmless because we force each rise in label to disk before transferring the data that caused the rise. Thus labels are robust against crashes; they are at risk only to wild scribbling in a file system, as by a berserk disk controller.

Direct labeling of files is not a panacea. Removable file systems can be moved only among systems that agree on label encodings. Moreover, labels will probably have to be translated between remotely communicating systems; but this is also true with indirect labels. (Indirect labels have been used in System V, where reason 5 does not pertain, and inode compatibility is of concern.[7, 8])

These considerations do not apply to labels that reside in main memory, where indirection is used. In system tables labels are shared via pointers to save space, to speed copying, and to speed comparisons. A further level of sharing guarantees that coupled labels, such as the labels on the two ends of a pipe or on a process and its associated process file vary together.

2.7. The floor, integrity labels

Biba pointed out that a secrecy lattice, where a highly cleared program can read almost any file and write almost none, is just the opposite of an 'integrity' lattice, where a highly trusted program can modify almost any file but can read almost none for fear of becoming contaminated with bad data.[9, 10] IX labels can be used in Biba fashion, by starting untrusted users high, and keeping trusted software low. Any change to trusted software caused by untrusted users will be reflected in a raised label. By running with a low ceiling, trusted applications may be immunized against using the corrupt software.

To get integrity and secrecy controls simultaneously, we simply regard some of the bits in a label as integrity bits, others as secrecy bits. All users log in at an administratively defined middle, or *floor* label, which has ones in its integrity bits and zeros in its secrecy bits. System source, utilities, and other important universally available files are labeled below the floor, typically at bottom.

In illustration, suppose that a project has one integrity bit and one secrecy bit. The floor label at which everybody—including project members—logs in is 10, with a 1 for low integrity and a 0 for low clearance. Project members are entitled to raise their ceiling to 11. Having begun at 10, their process labels can change to 11 but not to 00 or 01. The project administrator, who is entitled to 'downgrade' (i.e. upgrade in integrity) the integrity bit, may set the label on critical project data to 01. Should any ordinary project member succeed in writing the file, the integrity bit will change and the file label will become 11. The loss of integrity is thus recorded in the file label and optionally in a system audit trail (§5).

Marking data as high-integrity does not protect the data from compromise. It does, however, protect against false belief in the high integrity. By setting the ceiling to cut off access to low-integrity data, a high-integrity process may be immunized against unwittingly using corrupted data. Of course low-integrity processes may still use the corrupted data, but as their outputs would have low integrity anyway, the integrity policy has been respected. The only real hazard is denial of service to high-integrity processes.

New system code will usually be tested at a normal label, at or above the floor. It will be installed by downgrading the source code to a label below the floor, rechecking for accuracy, and compiling afresh. To be really careful, one would place the trusted computing base (§3.4) below everything else. Then no untrusted tools could be invoked inadvertently while working on trusted code.

3. PRIVILEGES

The data-flow policy, which covers normal operations, is insufficient for administrative purposes. For example, establishing a user session at a non-floor label may involve changing the process label down or the ceiling up, in contradiction to normal policy. Some other actions that violate the policy are document declassification, repair of multilevel file systems, opening of external media, and mounting of file systems.

Administrative actions that fall outside the data-flow policy are performed by privileged utilities, which are exempt in one way or another from kernel enforcement. As privileged code does not enjoy the guarantees of the policy, it must be written with great care to assure that it maintains intended security. Because such code is exempted from normal checks in the expectation that it will do no wrong, it is called 'trusted.' Privilege can be exercised only through trusted code.

Roughly speaking, the privilege mechanism partitions the supreme powers once accorded to the superuser. Superuser status itself is diminished. The superuser is fully bound by security labels and cannot ignore write permissions. To keep as much of the familiar UNIX semantics as possible and to avoid rewriting masses of code, we have retained the association of other special powers with the superuser. When these powers break the security rules, superuser status must be augmented by privilege.

3.1. Capabilities and licenses

Each privilege is governed by a one-bit *capability* and a one-bit *license.* A process possessing a capability has the actual right to exercise privilege. A process possessing a license has a potential right. Process licenses can be gained only with privilege, usually by application to the privilege server (§6.1), are inheritable across file execution (the *exec* system call), and may be relinquished at will. In effect licenses identify trusted users. Capabilities depend on the program being executed and are not inherited.

We denote the vector of capabilities of a process or file x by $Cap(x)$ and the vector of licenses by $Lic(x)$. In general, a process p executing file f will have a given capability if the file has the capability and the process is licensed for it. The vector of capabilities is computed by bitwise intersection:

$$Cap(p) = Cap(f) \cap Lic(p).$$

Other factors participate in the computation of capabilities. A program file has a license vector, $Lic(f)$, which may endow the executing process with capabilities regardless of the process's own licenses. Such a 'self-licensed' program gets power in much the same way as does a set-user-id program in ordinary UNIX systems, except that the power is not inherited on executing further programs. When file system FS is mounted it may be assigned privilege masks, $Cap(FS)$ and $Lic(FS)$ to exclude imports of spurious privilege, for example from untrusted remote machines in a network file system. Privilege masks may also be used to limit the locus of privilege to some well-managed region of the entire file space. If file f lives in file system $FS(f)$, the complete formula for process capabilities is

$$Cap(p) = Cap(f) \cap Cap(FS(f)) \cap (Lic(p) \cup (Lic(f) \cap Lic(FS(f)))).$$

Readers familiar with System V release 4ES will recognize homologies between IX capabilities and 4ES 'working privileges,' licenses and 'maximum privileges,' file capabilities and 'inherited privileges,' self-licenses and 'fixed privileges.'[11] Local control is somewhat stronger in System V, where a process can turn its working privileges on and off. Global control is somewhat stronger in IX, where licenses cannot propagate through execution of untrusted code. (The danger in licensing untrusted processes: a Trojan horse might be able to pass a license to a privileged program to do ill. To guard against this eventuality, a privileged program would have to assume the burden of verifying the legitimacy of each task it is asked to do.)

3.2. Specific privileges

Believing that too many privileges would be unmanageable, we have provided just six generic privileges, rather than a long list of specific privileges such as 'register new users,' 'repair file systems,' or 'downgrade files.' The existence of broad privileges does not automatically imply broad powers for a user who is authorized to exercise privilege. In practice, even trusted administrators are not granted active licenses. Instead, one applies to the privilege server (§6.1) to run each privileged command. If a user is authorized to run the command with the given arguments, the server grants the necessary licenses and executes the command. The six privileges, listed in roughly descending order of power, are

Set privileges. Change file capabilities and licenses.

Set licenses. Increase the license or ceiling of a process.

Extern. Mount file systems, change labels away from **no**, or change labels of external media.

Nocheck. Read or write data without regard to security label.

Control auditing. Adjust the intensity of auditing, nominate auditing files, or perform related actions.

Write uarea. Change system-maintained data such as userid, which may be read by other processes.

The last-mentioned privilege is more an artifact of UNIX than a corollary of the security mechanism (§7.2). Otherwise each privilege overrides an identifiable aspect of that mechanism.

3.3. Concerns in privileged code

Privileged processes in effect administer their own security policy, and thus must be written with the same care as the kernel itself. As an example, we consider the privilege 'nocheck,' which allows a process to circumvent automatic label checking on selected files. Nocheck is used in various trusted applications that have to read and write multilevel data. Among these applications are: checking, copying, and repairing multilevel file systems; managing a multilevel terminal over a single-level wire (§6.4); delivering multilevel mail; and providing a centralized service (e.g. password authentication, §6.3) for multilevel clients.

A nocheck process must take care to observe that file labels do not change underfoot in dangerous ways. Notification tools exist for this purpose. A new signal, SIGLAB, may be used to detect changes in labels of open files; and a special system call (*unsafe*) identifies which files are involved. It is up to the process, however, to decide whether each particular label change is safe or not.

3.4. The trusted computing base

To a first approximation, the kernel and privileged programs constitute a trusted computing base (TCB).[2] Nothing can join the TCB without the intervention of something that is already in it. Nothing in the TCB can change. The only way to modify a trusted program is to delete it from the TCB by removing its privilege, work on it, and then restore privilege. A paranoid protocol for modifying the TCB is given in §6.6.

Anything that prepares or verifies components of the TCB should either have its effects vetted by a TCB program or itself be in the TCB. For example, a compiler whose correctness is taken on faith should be in the TCB, while a compiler whose output can be verified by an independent TCB tool need not be. A part of the TCB that does not need privilege must be made immutable in some other way. One way is to give the program a dummy license, but no capability. Such a program is formally privileged, but can exercise no privileges. Another way is to protect the TCB by integrity labeling (§2.6).

4. PRIVATE PATHS

In some instances security may be compromised by simultaneous access of trusted or untrusted processes to one file. Consider, for example, the apparently simple matter of collecting a password at login time from a user at a physically secure port. Until authentication is complete the user's clearance is unknown, so the port cannot be honestly labeled at any ordinary security level. If the port were unclassified, other processes might be able to steal the password. If the port were highly classified, high data could be sent to it without the recipient having yet been authenticated. Thus to assure that passwords go only where they are supposed to, and that the port not be misused for other information, the port should be placed in a special dedicated state. As long as the port remains in this state no other process can read, write, or seek on any file descriptor associated with it.

A properly cautious user will demand assurance that the password is in fact being demanded by trusted code. Hence the special state of the port and the credentials of the communicating process should be discernible by the user as well (§6.4). We call the arrangement a *private path.*

Should other trustable communications processes be interposed in a private path, then the (two-way) assurances need to extend transitively to each segment of the channel. If the trust breaks at any point of the channel, data that started along the channel in trust must not be misdelivered, and data that is delivered in trust must not have been injected from an unintended source.

IX has two features for administering private paths: process-exclusive access and stream identifiers. These features are used mainly to insure the integrity of transactions performed by trusted processes.

4.1. Pex

A process can obtain a private path by asserting process exclusive, or *pexed,* access to a file or stream that is open in the process. Pex may be used to lock out interference when updating a file (§6.6). If one end of a pipe is pexed, the pipe cannot be used unless the other end is pexed, too. Moreover, the pexing protocol provides to the process at each end of the pipe unforgeable indicia of trust (userid and capabilities) about the other. The indicia provide the basis for extending trust transitively along a multistep private path.* When either process breaks trust by unpexing its end, its partner is prevented from blindly continuing as if nothing had happened: the pipe becomes unusable until unpexed at both ends.

Transmissions on a pexed file or stream are understood to be immune to eavesdropping or injection of signals. Since such assurance is not necessarily available for external media, external media may be conditioned to allow or disallow pexing. Thus pexing might be denied on a communications port to an untrusted computer and permitted on a hard-wired line to a trusted terminal in a secured area.

Although pex is used almost entirely by privileged processes, it is available to any process. Herein lies a risk. By pexing a file, a malicious process can deny its use to others. The process can be forced to let go only by killing it.

4.2. Stream identifier

The destination of a file, particularly if that file connects to the outside world, may determine how it can be used. We have just explained one determining condition, the ability to use the stream as a private path. Others are the identity of the port's user, the tokens or passwords the user has presented, the stream's network source, etc. For recording such facts about a port, IX allows an arbitrary string, called a *stream identifier,* to be associated with any UNIX stream. Settable only with privilege, the stream identifier serves as a trustable repository of descriptive information. The stream identifier is usually set by the login program and may be augmented by the session supervisor (§6.2).

Pexing and stream identifiers associate security-related properties with channels, not with subjects or objects. In a communications-based system, the relationship of a stream to a subject may be remote indeed. What if the stream leads to a multiplexer process, to an encrypter, to a data link, to another computer, to a decrypter, to a demultiplexer, to ...? It does not make sense to consider all stages in the path to be independent subjects ruled by the security policy. A multilevel demultiplexer, for example, must be able to violate the letter of the policy, while being trusted to carry out the intent. In order to do that it needs to know the

* Thus realizing something like Boebert's 'assured pipeline.'[12]

security requirements of the data it is handling, even though it has no need to know about the origins of that data. Streams, not users, are what computer processes deal with. As streams become more complicated, it becomes increasingly important to characterize their properties, which may be only weakly correlated with the properties of the agents they connect.

5. AUDITING

Audit records are kept with selectable intensity by the kernel on special audit devices. Audit devices are also available for record-keeping by security-critical programs, such as the privilege server (§6.1) or the login program. An audit device is associated with an ordinary UNIX file by a privileged system call. The file itself becomes unreadable without privilege, but is universally writable. For maximum security, an audit device may be assigned to a data link to an off-line repository.

System events are classified into 11 independently auditable categories, selectable by one of four audit masks. All processes are audited at the level specified by a global mask. Individual files and processes may be invisibly *poisoned* to add auditing as specified by one of the other masks. A process that touches a poisoned file becomes poisoned. Poisoning is cumulative and inherited across processes. Poisoning allows the audit level to be cranked up in response to a sensitive or suspicious action or for selected users.

6. SECURITY-RELATED PROGRAMS

This section describes some distinctive and important programs in the TCB.

6.1. Privilege server

The privilege server hands out the exact privilege needed to execute administrative tasks. It is guided by a data base of privileged commands and conditions for their execution. On request for a particular privileged action, the server checks authorization for that action and invokes the action under carefully controlled conditions. The privilege server imposes more structure on authorizations than do homologous programs in other UNIX systems. As a result, IX can enforce a notion of a well formed edit of the privilege data base.

Authority may be made delegable, so suborganizations can edit parts of the privilege data base controlled by their superior organizations, without intervention by an omnipotent system administrator. In particular, it is possible to give a particular user complete administrative control over a particular bit (or security category) in the label space.

The privilege server's data base consists of a tree of authorizations and a set of action rules.[5] The nodes in the authorization tree are labeled with triples of form (*pathname*, *access_predicate*, *rights*). The pathname names the tree node and the access predicate is built with AND and OR out of these kinds of atoms:

A regular expression for the client's login name.
A password demand for the password server to verify.
A regular expression for the stream identifier of the client's standard input.

Rights are a set of capabilities that access to the node confers. The simplest rights are plain identifiers. Other rights are identifiers that bear limiting values that confine the right to part of some lattice. Currently we support these nontrivial lattices:

The lattice of labels.
The lattice of regular languages ordered by inclusion.
The lattice of subsets of privileges.

In mathematical terms, *rights* names an element of the Cartesian product of as many lattices as there are distinct identifiers (with plain identifiers understood as tops of two-element lattices). Rights are monotone in the authorization tree: if node x is above (closer to the root than) node y, then the rights of x dominate those of y. By convention, the root of the tree has all rights. The root of the authorization tree represents the undiluted power of the TCB, and the various subnodes represent various limited ranges of powers, suitable for granting to various users.

In response to a user's request, which looks like a shell command, the privilege server searches for nodes whose access predicates are satisfied. The user is provisionally granted the set of corresponding rights. Then the request is compared with the action rules, which are triples of form (*template*, *need*, *action*):

> The *template* is a regular expression describing the request.
> The *need* defines the minimum rights necessary.
> The *action* tells what to do.

If some provisional rights dominate the *need*, then the *action* is carried out. Both the need and the action can contain parameters substituted from substring matches in template.

Before execution, the proposed action is presented under cover of pex (§4.1) for the user's confirmation. Requests placed from an insecure terminal or from a remote computer will be rejected because the pex will fail. However, a request can safely be issued through untrusted software, in particular the shell, because the user gets to vet and confirm the action over a private path to the privilege server.

Usually the action specifies execution of a particular program, with process license and ceiling set to particular values. The action may also specify an edit of a named subtree of the authorization tree, always preserving the monotonicity of the authorization tree. Authority is delegable by granting the right to edit a subtree.

When an action has been approved by both the privilege server and the user, it is executed in a paranoid, context-independent way. There are no environment variables. The invocation is direct, unmediated by a shell. The current directory is set to a 'black hole' labeled **no**; thus files can only be referred to by absolute path names.

For example, suppose the authorization tree has a node

pathname:	/admin/networks/internet
access predicate:	SRC(/dev/console)
rights:	netoper

and immediately superior node

pathname:	/admin/networks
access predicate:	ID(ches) & PW(ches)
rights:	netadmin | netoper

(Monotonicity requires at least the right netoper, since access to the superior node must imply access to the inferior node.) The first access predicate grants rights to privilege requests that originate at /dev/console; the second to a user with a certain login name who can present a password appropriate to that name. If one of the action rules is

template:	tcpgateway
need:	netoper
action:	PRIV(x), EXEC(/lib/tcp/tcpgateway)

a request for the privileged action tcpgateway will be admitted if the source is /dev/console or if the login name ches can be confirmed. Only then will the action be performed, executing the desired program with external-medium (x) privilege.

The project administrator in the example in §2.7 might be granted access to a node with right downgrade(projectbit), where projectbit names a part of the lattice of labels assigned to the project. The downgrade request would need downgrade(arg1) privilege, where arg1 refers to the first argument of the request and must name a label. Then the administrator could use the following command to declassify a report issued by the project.

```
priv downgrade projectbit report
```

Although the administrator has downgrade privilege, the privilege is confined to the one project.

The right to edit the privilege data base must be carefully controlled, for whoever can edit the root of the authorization tree holds an ultimate key to the system. To keep that right from being exercised unilaterally, it may be protected, for example, by requiring two or more passwords, which are known by different people. Beyond the privilege server, there is one more source of ultimate power—single-user mode at the main console (§6.5), access to which must be controlled by physical security measures.

6.2. Session supervisor

The session supervisor handles requests for sessions with a different label, much as the standard *su* command does for sessions with a different userid. It accepts requests to change the label of the terminal (standard input) and the process ceiling. After verifying the user's clearance and the suitability of the particular port for traffic at the desired label, the session supervisor resets the label of the standard input to the desired level, adjusts the process ceiling, and invokes a shell. At the end of the session, the supervisor restores the label of the standard input and exits. Label inequalities prevent misuse of the terminal by processes that survive across either label change.

6.3. Password server

Ordinary UNIX passwords are open to compromise from eavesdropping. As a countermeasure, IX supports cryptographic protocols. Unfortunately such protocols, unlike classical UNIX passwords, depend on the system knowing secret keys, which must be very closely held. Password checking may have to be done at any security level, so the password file can't be protected simply by giving it a high label. For these reasons IX has a privileged server that collects passwords from the user and verifies them. An incidental benefit is that new authentication algorithms can be adopted easily, for only the password server needs to know them.

A client program (such as *login* or *su*), which needs to verify the claimed identity of the party at the far end of a file descriptor, establishes a connection to the password server through a pipe mounted in the file system.[13] The connection is assured by pex (§4.1); if pexing fails, the identity claim is denied. The client then sends to the server the claimed identity and the file descriptor.

The server attempts to verify the claimed identity by using one of two authentication algorithms. The user is asked to provide either a classical UNIX password or a response from a cryptographic pocket calculator. If the user's file descriptor cannot be pexed, meaning that the channel cannot be trusted to keep passwords secret, only a cryptographic response is accepted. After the user's reply is collected and checked, the server reports success or failure to the client and drops off the line.

Because only one program needs access to the password file, that file could be kept off line for safety. The password server would then become a stub that observes the password protocol and maintains exclusive access to the off-line connection.

For a system administrator engaged in a spurt of privileged actions, it would be tedious to reauthenticate for every action. And it would be dangerous to use a privileged shell, for such a blanket grant of privilege raises the possibility that the privilege may be exercised in unintended ways. Instead the session supervisor can create special sessions, wherein the administrator's I/O stream is tagged (in the stream identifier, §4.2) to tell what authenticating checks have been successfully passed. The password server uses the information to short-circuit its authenticating protocols. (Pex-protected user confirmation is still required, however, to prevent untrusted software from initiating unintended privileged actions.) During such a session, the administrator can invoke the privilege server without having to present a password at each invocation.

6.4. Multilevel windows

To illustrate the use of private paths, we consider how multiple asynchronous windows are managed on a bitmapped terminal, the Teletype 5620. Since the terminal has no hardware memory protection, all code in the terminal must be trusted if it is to run as a multilevel device. Moreover that code must adhere to the label policy on data transfers (cut and paste) among windows.

All communication with the terminal passes through a host multiplexer program, *mux.*[14] Normally a process group, comprising a shell and programs that it invokes, has its standard output connected to *mux* by a pipe, one process group and pipe per window. To the processes the pipe looks like a conventional terminal, running at some fixed label. *Mux* itself must run with privilege to be able to deal with differently labeled windows.

Interesting things happen when a privileged process is invoked in a window. For example, the UNIX super-user command, *su*, demands a password. (To be precise, *su* calls on the password server to demand the password.) The command pexes the pipe to *mux*, which reciprocates by pexing the other end and extends the private path to the window code in the terminal. In turn a characteristic pattern is displayed in the window to announce the private path. Thus the user is assured that the password will be treated safely. In this respect, the more complex multilevel terminal offers an advantage over a simple dumb terminal, which is unable to deliver such out-of-band assurance.

To work at a different label in a window, one invokes the session command in the host. After determining that the user is properly cleared, the session command sets the label on its input pipe. The pipe leads to *mux,* which detects the change and passes it on to the window. Unless the new label dominates the old, the contents of the window are erased. A new shell is spawned for the duration of the session. Although an old shell and possibly other processes are also attached to the pipe, no attempt to access the pipe from these processes can violate the label policy. When the new shell exits at the end of the session, the pipe label is restored, the label change is propagated to the terminal, and the window's memory is erased if necessary.

6.5. Nosh, a believable shell

The customary UNIX shell programs are extremely complex. Their semantics are incompletely defined and their behavior depends heavily on the environment. They can all too easily be made to do the unexpected, and hence are dangerous instruments for system administration.

For critical uses in IX we have written *nosh,* a no-surprise shell. *Nosh* is invoked for the single-user maintenance shell that comes up with all privileges at system boot and for the customary boot script (*rc*). It is also used in sessions labeled below the system floor, where system updating is done. By all measures *nosh* is less than 6% the size of the classical Bourne shell; and its 400 lines of source are coded in a direct, understandable style. It is most easily characterized by the features it lacks. *Nosh* has

no path search; all commands begin with / or ./
no variables
no environment
no user profile
no aliases or defined functions
no redirection of I/O other than standard output
no pipelines (|)
no compound commands (if, for, case, &&, ...)
no background commands (&) or job control
no manipulation of signals
no file name generation (* or ?)
no command substitution (‘...‘)
no history
no mail notification
no arithmetic or test commands

When licensed for privileges, as it is in single-user maintenance mode, *nosh* refrains from passing licenses to other commands except by specific request on each command. *Nosh* retains the customary shell flag −e, which causes an immediate exit when any command fails; this feature is important for the boot-time shell script (/etc/rc).

6.6. Pcopy, software installation

A privileged file cannot be changed (§3.4). Thus to copy a privileged file, including its licenses and capabilities, one must first copy it to an unprivileged place and then mark the copy privileged. In the meantime, the copy may be open to meddling. *Pcopy* does the job 'atomically' under cover of pex.

In a system where powers are strictly separated between a system administrator (sysadmin) and a security administrator (secadmin), the job of replacing a privileged program P might be divided between them according to the following scenario. Steps 1 to 5 may take considerable time. Steps 6 to step 8, however, should be done quickly, because program P is unavailable during that interval. *Pcopy* is used to assure the integrity of the copy. Except where noted otherwise, each step is mediated by the privilege server, whose tables provide just enough privilege to each administrator. Neither can do the job alone.

1. Sysadmin compiles, or receives from some official source, new code P'. No privilege needed.

2. Sysadmin protects P' from modification by giving it a dummy license (§3.4). Secadmin does not have the right to do this.

3. Sysadmin checks the contents of P', perhaps by code testing, decompiling, or comparing against a reference copy. No privilege needed.

4. Secadmin adds a dummy license to P'. Sysadmin does not have the right to do this.

5. Secadmin checks the contents of P', perhaps by cryptographic certification or virus scan. No privilege needed.

6. Sysadmin removes the privilege from P and uses *pcopy* to copy P' and the dummy licenses to P.

7. Secadmin compares the freshly written P to the original P'. No privilege needed.

8. Secadmin sets the final privilege on P and removes the dummy licenses. To enforce the separation of powers, this step should be conditional on the proper dummy licenses being in place.

9. Either administrator removes the dummy licenses from P'.

7. SYSTEM MATTERS

7.1. Efficiency

The efficiency of IX relative to the underlying 10th Edition system varies depending on how frequently label computations must be made. Listing a directory, which requires a label check for each file, incurs a 15% time penalty. The overhead on creating a file in the current directory, writing one byte in it, and destroying it is 20%. Long path names, which require a label comparison at every intermediate directory, make things worse. An inefficiently coded file-tree walk drags miserably.

For reading and writing, however, where the caching of label checks comes into play, the efficiency story is different. The overhead is immeasurably small for typical file-processing programs working on files more than a few disk blocks long.

7.2. Special UNIX considerations

Most of the security facilities of IX are generic and reflect the policy, not the system. A few, however, deal with features peculiar to UNIX.

Blind directory. To preserve the portability of UNIX code, the common temporary directory, /tmp, had to be kept, and somehow be usable by processes of different labels. After considering, and even implementing, various alternatives, we settled on the simple notion of a blind directory. A blind directory cannot be read; hence no data flows can occur through it and its label does not matter. A blind directory admits covert channels, however. For example, a low process can learn via error returns whether the directory contains files by known names, regardless of the files' labels. Or a low process can create a collection of links and watch link counts change as a high process unlinks them.

A somewhat stronger 'doubly blind' technique for concealing temporary files was also implemented, but was dropped because it changed the system interface in a way that affected every program that created a file in the temporary directory. In this scheme, the system generated a random name for each file created in the blind directory and returned the name to the creator.

Seek pointer. Two processes that through inheritance share an open file also share the seek pointer on that file. By using the UNIX system call *lseek* in one process to set the seek pointer and in another to read it, a remarkably high interprocess bandwidth can be attained in classic UNIX systems. As a medium of data flow, seek pointers must be labeled. The labels of seek pointers should not be tied to either process or file labels, for that would unnecessarily force identity of labels in many instances. Hence each seek pointer has a label of its own.

The standard UNIX system call *lseek,* by virtue of both writing and reading the seek pointer, has the side effect of forcing the labels of the file and the process to be equal in most cases. For programs that wish to circumvent this unfortunate property, we resuscitated an ancient pair of system calls, *seek* and *tell,* which separate the functions.

User area. Associated with each UNIX process is a collection of descriptive information, which includes user number, login name, permission mask (*umask*), permission group, process ceiling, and other items. As the listed items can be set and read by system calls and are inherited by successor processes, they provide data-flow channels between processes, which could be exploited in connection with the drop-on-exec convention (§2.3). We have interdicted most of them by defining a peculiar privilege for setting user-area items. Most of the items need super-user status anyway; the need for privilege in connection with the user area affects only administration, not ordinary usage.

Different mechanisms are used to protect two user-area items settable by ordinary users, the permission mask and the process ceiling (which can be revised downward). To prevent downward data flow through the permission mask, it is set to a default value when drop-on-exec occurs. To prevent downward flow through the ceiling, the ceiling itself has a label that is set whenever the ceiling changes. A privileged ceiling change sets the label on the ceiling to bottom; other changes set it to the process label.

Of the three mechanisms for protecting user-area data (a special privilege, censoring, and giving each item its own label) the most straightforward is the last. But it is not clear what restoring force, analogous to dropping the item's label when the item is set with privilege, can be provided for labels on items other than the ceiling.

7.3. System changes

IX is based on the 10th Edition research UNIX system.[15] It has the following special system calls over and above those of the 10th Edition:

get/set file label and privileges
get/set process label and privileges
get file system ceiling
mount, setting file system ceiling
control auditing
control nocheck privilege on individual open files
identify open files with changed labels
seek and tell (§7.2)
I/O controls for pexing (§4.1)
I/O controls for stream identifiers (§4.2)

A few rare system calls have been abolished. One is *chroot,* whose potential for mischief overtaxed our understanding. The other two, *setgroups* and *getgroups,* entailed a user-area covert channel of enormous bandwidth.

The kernel changes were implemented in about 1300 lines of alterations to normal 10th Edition source, plus about 2100 lines of new modules. The total kernel source of under 33,000 lines handles the usual 10th Edition system calls (no System-V style IPC or shared memory), console, disks, Datakit networking, and remote file systems. Further device drivers could be taken over verbatim from the 10th Edition.

Some extra kernel memory is required for labels, but sharing (§2.6) keeps the total low even though hundreds of 62-byte labels may be in use at one time. The data structure for maintaining coherence of the cached checks (§2.5) is another matter. In a system configured for 300 processes, this data structure occupies about 150K bytes including file descriptors, which cannot be swapped with their processes as had been possible before. The data structure is coded expansively, with several fixed-size tables and lots of 32-bit pointers. Although its storage requirements could be reduced by a significant factor, we have not bothered to optimize it, for it works adequately.

The layout of file systems was changed to accommodate labels.

About ten maintenance utilities (e.g. *fsck* and *mount*) had to be adapted to the new file system and kernel layouts. A handful of security-related utilities (e.g. *login* and *su*) had to be modified to use the new security mechanisms.

Some new utilities have been written. Besides those described in §6, there are programs to set and retrieve labels, handle multilevel mail, and administer auditing. The new utilities comprise about 6000 lines of code, of which the privilege server accounts for nearly half.

7.4. Contrasts with Orange Book UNIX systems

The 'tradition' in our title refers to that of simplicity and generality as established in early research systems. In commercial systems, 'tradition' connotes also burdens of history and market perceptions. We are fortunate to have been able to follow the former tradition, which allowed us to value coherence before compatibility and marketing demands.

Administrative tools. As yet, IX has no tools for handling labels symbolically, translating them on passing across machine boundaries, and so forth. Nor has it any significant tools for analyzing audit records.

Labeling output. IX does not provide visible labels on a windowed terminal or mandatorily place labels in printed output. Such labeling would be easy to enforce, because external devices can be reached only with mediation of privilege. Indeed, for windowed terminals, *mux* and the *layers* program in System V/MLS,[16] were built on the same basic model at the same time by a group with whom we were in touch. *Layers* handles the labeling with no difficulty.

Temporary directory Blind directories in IX are somewhat more open and more vulnerable to tampering than the 'multilevel directories' of System V,[7, 8] or the multilevel cloned copies of LOCK/ix,[17] neither of which is easily adaptable to work with dynamic labels.

Access control lists. We deliberately chose not to support discretionary access control lists (ACLs), which the Orange Book effectively requires. Their implementation poses no research challenge. More importantly, we have little faith in the protective value of such a complex, structureless mechanism, which must be even harder to keep in order than are ordinary UNIX permissions.

Covert channels. We believe that covert channels between deliberately cooperating processes are not a significant concern in most commercial applications. Accordingly IX admits covert channels much wider than the Orange Book contemplates. Some are a natural corollary of dynamic labels. The widest channels arise from the drop-on-exec convention (§2.3), the value of which is open to question. If that convention were abandoned in favor of some other mechanism for obtaining processes with low labels, many covert channels would vanish.

Trustable shell. The extreme programmability and the complex semantics of UNIX shells makes them dangerous. You cannot tell what a shell script does by looking at it, because the shell's behavior is so crucially dependent on its environment; administrative shell scripts are accidents waiting to happen. The availability of *nosh* allows us to exclude ordinary shells from the TCB. We thereby sacrifice some flexibility, but less than one might expect. For example, with the shell unable to generate file names, we can clean a directory by executing `rm −r .`, instead of the more usual `rm *`.

Private path. Private paths with two-way confirmation are essential to building virtual trusted systems, such as the *mux* window manager, on top of UNIX. The pex idea should generalize to multiple trusted computers, with pex-protected paths crossing machine boundaries by cryptographic means if necessary.[5]

We are not aware of any exact analogue of pex in other systems. Locking primitives in some systems achieve the privacy of pex, on files if not on IPC channels. But the mutual confirmation aspect of pex on pipes, which is important in client/server architectures, seems to be new. Neither 'assured paths' in LOCK[12] nor the ability to test privacy in System V/MLS[8] provide mutual confirmation.

IX has no exact Orange Book 'trusted path' facility, whereby a magic signal gets a direct connection to the TCB. Pex on a *mux* terminal (§6.4) comes close, but it is still wise to cycle the power off and on to refresh a terminal that others have been using.

Privilege. The privilege server's data base imposes structure on the pattern of authorization in the system and permits the delegation of authority. Corresponding programs in other systems work from flat data bases with entries equivalent to (*User*, *Added_priv*, *Program*), which allow specific users to run specific programs with specific privileges.* The structure provides a notion of 'coherent state' or 'legal edit', which are missing from a flat data bases, where he who edits the data base is king.

8. CONCLUSIONS

Although IX has run for several years as a full-featured UNIX system, it has not stood the test of abuse outside the laboratory. Nevertheless it did enjoy an early success stopping a virus. Infected code was received over an unclassified data connection, and from it the virus propagated among user programs, including some run by the superuser. It could not, however, propagate into trusted code; and it revealed itself by attempting to do so. Thus the system worked exactly as intended. The same virus infested other UNIX systems on the local network without being detected.[18]

Dynamic labels are an attractive model for mandatory security. They serve well for preventing unauthorized access and accidental disclosure, though not for interdicting covert channels. They are well matched to multilevel applications, such as multilevel windowed terminals. The uniformity of the model, with checking at every transaction, fosters a coherent implementation that works the same way across the spectrum of devices and system features. As a result, much of the work of label checking is expressible in tables (§2.5).

Label policies on the IX model should be useful in settings where several users work on several projects, with varying patterns of sharing. For example, in educational institutions, secrecy labels could enforce separation among drafts of examinations, grade books, and students. In medical labs, secrecy labels could isolate personal information about experimental subjects. Integrity labels could help distinguish maintenance access from everyday use, even when users and maintainers are the same people. IX labels should be adequate in many classified settings with need-to-know rules or concerns about inadvertent admixture of data; the potential for covert channels may be the least of one's worries about spying.[19]

A potential problem with dynamic labels is label creep. High data mistakenly written into a low file contaminates that file irrevocably. (Bell-LaPadula systems have a complementary problem: you can't write high data into a low file even when you want to.) The most dramatic problem is that of tar-baby files (§2.4). We believe, however, that label creep is a manageable problem. The alternative is overclassification, especially of compartmented data.[19] For example, an intelligence analyst looking at data from various sources in a Bell-LaPadula system will naturally choose a label at the upper bound of all the data. Outputs then will appear to belong to all compartments rather than just the compartments of the contributing inputs. This ought to be avoidable at a multilevel IX terminal. Unfortunately, however, we implemented each *mux* window as a virtual terminal bearing a single label just as an ordinary device file does. Thus the label of the keyboard is tied to the label of the highest output that the window can receive, and the host processes will be forced high regardless of what data is being examined. A plausible, and not too difficult, extension of *mux* would be to provide finer-grained control somewhat in the spirit of a 'compartmented mode workstation' from IBM.[20] In that system, which has dynamic labels and per-file ceilings, input labels are divorced from output labels and in some cases labels are kept to the granularity of a byte.

The mechanisms for stream security and structured privilege are largely concerned with assuring integrity, by establishing mutual confidence among actors and by guiding behavior in authorized patterns. These notions are important for safe and sound operations in general, whether or not security labeling is in force. Private paths are broadly applicable. Any UNIX system could benefit from them. Pexing is important for client/server applications and for multiplexed use of channels. Stream identifiers, or some equivalent way to attach characterizing information to channels, help considerably in safe use of heterogeneous networks.

---

* The UNIX login program is an example, where *Program* is the user's login shell, *Added_priv* is given by the user- and group-id fields, and *User* is given by the user name and encrypted password fields.

The privilege server, though somewhat complicated, should be useful in classical UNIX or almost any system with security features. In particular, it could be used to realize security models, such as Clark-Wilson, which have been recommended for commercial applications.[21] These models are more concerned with who can do what than with who can see what. The privilege server's authorization tree makes clear where rights originate and how they may be propagated. It corresponds well to the way organizations work by delegating and separating powers. It is a complement to, not a competitor for, identification and authentication facilities such as Kerberos.[22]

IX integrates mandatory controls with the UNIX system in a way that should meet most needs for confidentiality. Compact and coherent, IX has several unique features that engender trust in its behavior: continuously checked dynamic labels, structured management of privilege, and private paths.

ACKNOWLEDGMENTS

In building IX, we were fortunate to start from the 10th Edition research UNIX system, the structure of which had been carefully cultivated by D. M. Ritchie, D. L. Presotto, and N. Wilson. D. L. Presotto and A. G. Hume cheerfully modified software on which IX depended. K. L. Thompson, F. T. Grampp, and L. A. Wehr influenced major design decisions. Through J. D. Weiss and B. Smith-Thomas we kept in touch with the design of the B1-level System V/MLS; and through L. A. Wehr, D. Bendet, and G. B. Green with the B2-level System V release 4ES. Other technical contacts for those systems included C. W. Flink, T. F. Houghton, W. J. Leighton, and T. L. Vaden. C. Mumford of the Joel Chandler Harris Foundation gave enthusiastic help with reference material.

References

1. D. E. Bell and L. J. LaPadula, "Secure computer systems: mathematical foundations and model," M74-244, MITRE Corp. (May, 1973).
2. *Department of Defense Trusted Computer System Evaluation Criteria,* Department of Defense Computer Security Center, Fort George G. Meade, MD 20755 (15 August, 1983).
3. M. D. McIlroy and J. A. Reeds, "Multilevel security with fewer fetters," *Proc. Spring 1988 EUUG Conf.,* pp. 117-122, European UNIX Users Group, London (April 1988).
4. M. D. McIlroy and J. A. Reeds, "Design of IX, a multilevel secure UNIX system," CSTR #163, AT&T Bell Laboratories (December 1991).
5. J. Reeds, "Secure IX network" in *Cryptography and Distributed Computing,* ed. J. Feigenbaum and M. Merritt, AMS/ACM Series in Discrete Mathematics and Theoretical Computer Science (1991).
6. J. C. Harris, "Brer Rabbit, Brer Fox, and the Tar-baby," *Atlanta Constitution* (November 15, 1879).
7. D. Bendet, J. Ferrigno, G. B. Green, M. Hondo, E. Lund, and C. A. Salemi, "Challenges of trust: enhanced security for UNIX System V," *Proceedings, Winter Uniforum Conference* (1989).
8. C. W. Flink and J. D. Weiss, "System V/MLS labeling and mandatory policy alternatives," *AT&T Tech. J.,* 67, pp. 53-64 (1988).
9. D. E. R. Denning, *Cryptography and Data Security,* Addison-Wesley, Reading, MA (1982).
10. K. J. Biba, "Integrity considerations for secure computer systems," ESD-TR-76-372, USAF Electronic Systems Division, Bedford MA (April, 1977).
11. UNIX System Laboratories, *UNIX System V Release 4.1 B2 Enhanced Security User's Guide* (1991).
12. W. E. Boebert and R. Y. Kain, "A practical alternative to hierarchical integrity policies" in *Lock: Selected Papers, 1985-1988,* pp. 79-96, Secure Computing Technology Center 2855 Anthony Lane South, Suite 130, St. Anthony MN (1988).
13. D. L. Presotto and D. M. Ritchie, "Interprocess communication in the ninth edition UNIX system," *Software—Practice and Experience,* 20, S1, pp. S1/3-S1/17 (June, 1990).
14. R. Pike, "The Blit: a multiplexed graphics terminal," *Bell Laboratories Tech. J.,* 63, pp. 1607-1631 (1984).

15. AT&T Bell Laboratories Computing Science Research Center, *UNIX Research System Programmer's Manual,* 1, Saunders, Philadelphia (1990).

16. B. Smith-Thomas, "Secure multi-level windowing in a B1 certifiable secure UNIX operating system," *Winter 1989 Usenix Technical Conference,* pp. 429-439, Usenix Association, San Diego (1989).

17. Mark A. Schaffer and Geoff Walsh, "LOCK/ix: on implementing UNIX on the Lock TCB" in *Lock: Selected Papers, 1985-1988,* pp. 194-204, Secure Computing Technology Center 2855 Anthony Lane South, Suite 130, St. Anthony MN (1988).

18. Tom Duff, "Experience with viruses on UNIX systems," *Computing Systems,* 2, pp. 155-171 (1989).

19. J. P. L. Woodward, "Exploiting the dual nature of sensitivity labels," *Proceedings, Symposium on Security and Privacy,* pp. 23-30, IEEE, Oakland (1987).

20. M. E. Carson, J. Liang, G. L. Luckenbaugh, and D. H. Yakov, "Secure windows for UNIX," *Winter 1989 Usenix Technical Conference,* pp. 441-455, Usenix Association, San Diego (1989).

21. D. D. Clark and D. R. Wilson, "A comparison of commercial and military computer security policies," *Proceedings, Symposium on Security and Privacy,* pp. 184-194, IEEE Computer Society, Oakland (April, 1987).

22. J. Steiner, C. Neuman, and J. I. Schiller, "Kerberos: an authentication service for open network systems," *Winter Conference Proceedings,* Usenix Association, Dallas (February, 1988).