AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Computing Science Technical Report No. 163

The IX Multilevel-Secure UNIX System

James A. Reeds
M. Douglas McIlroy

January 1992

The IX Multilevel-Secure UNIX System

J. A. Reeds

M. D. McIlroy

ABSTRACT

A collection of papers about the IX system, a simple but comprehensive multilevel-secure operating system with mandatory access control, based on the research v10 UNIX® system.

The IX security model centers on processes and files or channels (not on ‘‘subjects’’ and ‘‘objects’’). The system calculates security-classification labels dynamically, so that outputs are classified as highly as the inputs from which they were derived. The label mechanism is *mandatory*; not even the superuser can subvert it.

A structured privilege mechanism allows system and security administrators to bend the rules in an orderly way for purposes such as maintenance or document declassification. Privilege may be suballocated in parts of the label space so that projects may administer their own security.

A private-channel mechanism guarantees freedom from eavesdropping or spoofing for communications among trusted processes and for special communications, such as password dialogs, with external sources.

The papers in the collection are

Multilevel Security in the UNIX Tradition. An overview of the IX system and important utilities. 19 pages.

The Design of IX. Detailed specification of the security behavior of the kernel. 32 pages.

A Tour of IX. Some examples of the use of security labels and of privilege in IX. 11 pages.

Multilevel Windows on a Single-Level Terminal. The workings of *mux*, a windowed-terminal handler, when it is possible for run differently classified sessions in different windows. 3 pages.

Secure IX Network. A discussion of the major security features of IX and how they could be extended to a network of secure computers. 8 pages.

Appendix.

Glossary. The jargon of IX that differs from that of UNIX. 2 pages.

Manual Pages. Features peculiar to IX described in the classical UNIX style. 50 pages.

Multilevel Security in the UNIX Tradition

M. D. McIlroy

J. A. Reeds

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

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.[3-5]

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.

1.2. Problems

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 $L(x) \leq L(y)$ 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 $L(x) \leq L(y)$. Moreover, if the causative process is p, then $L(y) \leq C(p)$. If x (or y) belongs to file system z, then $L(x) \leq C(z)$ (or $L(y) \leq C(z)$).

Data also flows in time. In general the label of any memory x must satisfy $L(x, t_1) \leq L(x, t_2)$ for $t_1 \leq t_2$. The ceiling of an unprivileged process must decrease monotonically with time: $C(p, t_2) \leq C(p, t_1)$ for $t_1 \leq t_2$. 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

If an uncleared 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, by using a command like*

```
pr <low.doc | lp
```

Drop-on-exec entails covert channels, most notably through the identity of open files.

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 touches 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 the 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

---

* The command prepares a file for printing, and pipes the result to a line printer. This example does not work literally, because a UNIX shell ordinarily supplies each program with a hidden argument, intended to be the program name. That argument, being received from a high process, contaminates the command and prevents it from running low. Thus we interpose a special command, `runlow`, which censors all arguments: `runlow pr <low.doc | runlow lp`

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

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.

At the same time, we accepted some difficulties. With fully labeled files, 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

---

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

the letter of the policy, while being trusted to carry out the intent. In order to do that it needs to know the 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($1)` privilege, where `$1` 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.

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 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 ([y8]). 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 specific UNIX features, and would likely differ in another operating system.

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 for many classified environments 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 largely illusory. 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 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-

---

* 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.

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] Bell, D. E. and LaPadula, L. J., ‘‘Secure computer systems: mathematical foundations and model,’’ M74-244, MITRE Corp. (May, 1973).

[2] Department of Defense Computer Security Center, *Department of Defense Trusted Computer System Evaluation Criteria*, Fort George G. Meade, MD 20755 (15 August, 1983).

[3] McIlroy, M. D. and Reeds, J. A., ‘‘Multilevel security with fewer fetters,’’ *Proc. Spring 1988 EUUG Conf.*, pp. 117-122, European UNIX Users Group, London (April 1988). Also *Proc. UNIX Security Workshop*, pp. 24-31, Portland (August 1988).

[4] McIlroy, M. D. and Reeds, J. A., ‘‘Design of IX, a multilevel secure UNIX system,’’ CSTR #163, AT&T Bell Laboratories (December 1991).

[5] Reeds, J., ‘‘Secure IX network,’’ in *Cryptography and Distributed Computing*, Feigenbaum, J. and Merritt, M. (Eds.), AMS/ACM Series in Discrete Mathematics and Theoretical Computer Science (1991).

[6] Harris, J. C., ‘‘Brer Rabbit, Brer Fox, and the Tar-baby,’’ *Atlanta Constitution* (November 15, 1879). Republished in many collections of Harris stories as ‘‘The wonderful Tar-Baby story’’.

[7] Bendet, D., Ferrigno, J., Green, G. B., Hondo, M., Lund, E., and Salemi, C. A., ‘‘Challenges of trust: enhanced security for UNIX System V,’’ *Proceedings, Winter Uniforum Conference* (1989).

[8] Flink, C. W. and Weiss, J. D., ‘‘System V/MLS labeling and mandatory policy alternatives,’’ *AT&T Tech. J.* **67**, pp. 53-64 (). Also *Winter 1989 Usenix Technical Conference*, pp. 53-64, San Diego (1989).

[9] Denning, D. E. R., *Cryptography and Data Security*, Addison-Wesley, Reading, MA (1982).

[10] Biba, K. J., ‘‘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] Boebert, W. E. and Kain, R. Y., ‘‘A practical alternative to hierarchical integrity policies,’’ in *Lock: Selected Papers, 1985-1988*, Secure Computing Technology Center 2855 Anthony Lane South, Suite 130, St. Anthony MN (1988).

[13] Presotto, D. L. and Ritchie, D. M., ‘‘Interprocess communication in the ninth edition UNIX system,’’ *Software—Practice and Experience* **20**(S1) (June, 1990).

[14] Pike, R., ‘‘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*, Vol. 1, Saunders, Philadelphia (1990).

[16] Smith-Thomas, B., ‘‘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] Schaffer, M. A. and Walsh, G., ‘‘LOCK/ix: on implementing UNIX on the Lock TCB,’’ in *Lock: Selected Papers, 1985-1988*, Secure Computing Technology Center 2855 Anthony Lane South, Suite 130, St. Anthony MN (1988).

[18] Duff, T., ‘‘Experience with viruses on UNIX systems,’’ *Computing Systems* **2** (1989).

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

[20] Carson, M. E., Liang, J., Luckenbaugh, G. L., and Yakov, D. H., ‘‘Secure windows for UNIX,’’

Winter 1989 Usenix Technical Conference, pp. 441-455, Usenix Association, San Diego (1989).

[21] Clark, D. D. and Wilson, D. R., ‘‘A comparison of commercial and military computer security policies,’’ *Proceedings, Symposium on Security and Privacy*, IEEE Computer Society, Oakland (April, 1987).

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

The Design of IX

M. D. McIlroy

J. A. Reeds

ABSTRACT

The mandatory security behavior of the IX kernel is specified semiformally. The security policy and the label mechanisms and checks that implement the policy are given, as are arrangements for privilege, private paths, and auditing. The security behavior of special files and of all system calls, new and old, is described. Covert channels are illustrated.

1. Introduction

In a *multilevel secure* operating system all data files have security classification labels. *Mandatory controls* assure that no combination of computer programs may copy data from a file into another with a lesser label. Thus the normal flow of data is *up*: as data move from place to place, the classification label must not decrease as a result of negligent or unauthorized action. However, certain *privileged* programs are allowed to copy data *down*, that is, to declassify or *downgrade* data.

We describe a simple, but thorough, way to add mandatory controls to a UNIX® operating system without severely impairing the basic nature and usefulness of the system. This paper recounts the modifications to the calling interface to the system kernel. Using this kernel, we have built a multilevel secure system called IX, which includes tools for authentication, administration of privilege, safe networking to untrusted machines, and management of multilevel windowed terminals. Multilevel secure networking should fit well within the model.

Higher-level aspects of the IX system are described in the accompanying paper, ‘‘Multilevel security in the UNIX tradition.’’ Based on the Tenth Edition UNIX research system, commonly known as v10,[1] IX was built to experiment with new security mechanisms. It bears no direct relationship to security features in production systems from AT&T.[2, 3]

Section §2 covers the ideas, §3 gives details. The reader is expected to be fully familiar with UNIX system calls. The technical details are deliberately concise, as they are intended as an implementation reference. Additional expository material is relegated to fine print.

2. The model

Each file or process has a *label*, shared by all data in it. For technical reasons given in §3.5.10 and §3.6.7, seek pointers and ceilings, which are defined below, also have labels. The labels form a (slightly augmented) mathematical lattice, a structure rich enough to express both multilevel and categorical, or compartmented, classification systems. Whenever a system call causes a transfer of data, the labels are checked to ensure that data only flows up the lattice.

The security of data explicitly passed between labeled entities, in particular from process to file and vice versa, is safeguarded. Examples of such data are bytes transmitted by *read* and *write* system calls and bits set by *chmod*. Implicitly set inode data, such as modification times and link counts, are protected as far as possible without making the system unusable. Special consideration is given to external media such as terminals or tape drives, where authentication protocols may be required in order to determine proper labels. To reduce overhead in label checking we cache the results of label checks involved with file descriptors.

Other ways of communicating information, including but not limited to error returns from system calls, file change times, the identity of open files, and otherwise inferred knowledge (e.g. the Denning example in §3.2.6) we declare to be *covert channels.* Just which covert channels to leave unplugged we have decided by balancing risk versus utility and compatibility. At worst, covert channels of nontrivial bandwidth provide routes for leaks, not for burglaries. As extremely abnormal behavior is required to exploit them (see notes in §3.5.9), systematic use of covert channels should be easy to detect by auditing.

In effect we have divided information transfers into ''lawful'' transfers, which honor the Department of Defense ''Orange Book'' criteria,[4] and covert channels.

We attempt to minimize label inflation by keeping all processes and files at their minimum allowable labels as long as possible. A program's label may start low and drift up as necessary to a maximum value authorized for its user. The label rises only when needed to allow reading of inputs. When a running program's label rises, the labels of its output files may also rise correspondingly.

A few system programs must be exempt from the usual label checking described above. Such programs are trusted with special *privileges,* which give them the ability, for instance, to set the label on a user's terminal at login time, read foreign tapes, perform backups—and assign privilege. These privileges are zealously guarded: a program cannot pass its privileges to another and privileged programs cannot be modified.

Privileged processes, which have the right to break the rules, must know that they are doing so safely and are not allowing unwashed programs to piggyback on the privilege. For example, the login program, which authenticates a user's identity and then sets security labels accordingly, needs to receive the user's password via a path immune to eavesdropping by untrusted agents. (Security clearance is not at issue; no distinguishing security label to guarantee the user's privacy can be established before the user has been identified.)

To obtain a private path, such as that necessary for logging in, a process may assert process exclusive, or *pex,* access to any file or pipe. On a pipe, the processes at both ends are apprised of each other's trustedness. On an external connection, an associated *stream identifier* may be queried for other assurances, such as whether it is understood to be physically secure.

An audit mechanism records security-related events. Audit records are collected mandatorily, to an administratively determined level of detail. Extra audit records may be volunteered, typically by privileged programs, to capture data (e.g. password rejections) that happen outside the kernel.

In summary, IX has five major security mechanisms:

1. The usual UNIX permission scheme provides discretionary controls. The superuser can override permissions other than write permission.
2. The label scheme provides mandatory controls. Label inequalities are maintained regardless of user- or group-ids.
3. The privilege scheme guards the administration of the label scheme (and of itself).
4. Process exclusive streams allow private transmission of data among privileged processes and files.
5. Detailed auditing allows security surveillance and furnishes post-mortems in case of trouble.

Each process has a *ceiling,* a maximum label that it may read or write, first set when the user logs in. Ceilings prevent lowly users from injecting noise into high places. Ceilings also prevent lowly users from raising their processes to high levels where they might use covert channels to pry out secrets. Only with the (possibly unwitting) collusion of a (possibly duped) cleared user can covert channels be used to see above the ceiling.

Each system call is identified as a *read action,* a *write action,* or both, depending on the direction of data transfer. The destination of a read action is a process; the destination of a write action is a file. A security check is made at each system call. When a check is violated, the violation may sometimes be prevented by changing a label, otherwise the system call aborts. Thus, in the absence of privilege, the system obeys two golden rules, which are elaborated in §2.5.

Upward flow. Only upward data flow is permitted. If possible, the label of the destination of an action is raised to allow the action to proceed.

Impenetrable ceilings. A process label must stay under the process ceiling.

Each file system has a ceiling label, distinct from the labels of any file in it. No information with a label above the ceiling can be transferred to or from the file system. The ceiling, which may be understood as a process label on a virtual file manager, may be used to prevent import or export of sensitive labels via remote file systems or removable media. In addition, the file system ceiling may be used to deny privilege to executable files obtained from such sources.

We understand the system call *exec* to extinguish a process and make in its place a new *empty* process. To keep labels as low as possible, an empty process begins with a bottom label. If an empty process has arguments, its label may have to rise immediately to cover the label of its parent, the source of the arguments. It may have to rise further to cover the label of the initializing text file, and perhaps again to cover data that it reads.

2.1. Terminology

A *file* is anything that can have a file descriptor, or equivalently anything that can have an in-core inode: file system entries, pipes, and process images. An inode is deemed to be part of its file. An open file that is not a stream has a *seek pointer*. A file descriptor d names an association (p, s, f) between process, seek pointer, and file. Given d, the corresponding process is denoted $p(d)$, the file $f(d)$, and the seek pointer $s(d)$. The file system that file f resides in is denoted $FS(f)$. If f is one end of a pipe, f' is the other end. $L(f,t)$, $L(p,t)$, $L(s,t)$ are the labels of a file, a process, and a seek pointer respectively at time t; $C(p,t)$ and $C(FS,t)$ are the ceilings of a process and file system. When only one time is under consideration, t may be elided: $L(f)$, $L(p)$, $C(p)$, etc.

A *data flow* occurs when bits are copied from one place (process, file, seek pointer, uarea) to another. Such flows, caused by system calls, are effectively atomic and are serializable. A nonatomic data transfer in a very long *read* or *write* is considered to be several data flows. The residence of data in an entity (usually a process or file) also constitutes data flow, called a *persistent flow*, from the entity at one time to the entity at another time.

Transfer of information without direct replication of bits is not considered to be data flow. Most such transfers are sensed by error returns from system calls. Others are sensed by reading quantities that the system calculates: link counts, process numbers, resource levels, file access times, clock values, and so on.

Data flow from source x to destination y is symbolized $x \rightarrow y$.

The symbols := and = mean assignment and the equality predicate respectively.

☞ In the more formal parts of the paper fact is distinguished from supporting commentary, which looks like this.

2.1.1. Summary of notations

Notation	Meaning	Reference
f, r, w	file	
d	file descriptor	§2.1
p	current process	
q	process	
t	time	
s	seek pointer	§2.1
FS	file system	§2.1
x, y	labelable entity	
y	label yes	§2.2
n	label no	§2.2
$L(x,t)$, $L(x)$	label	§2.2
$C(x,t)$, $C(x)$	ceiling	§2.2
$\mathrm{Cap}_k(x,t)$, $\mathrm{Cap}_k(x)$	capability	§2.4.2
$\mathrm{Lic}_k(x,t)$, $\mathrm{Lic}_k(x)$	license	§2.4.2
$\mathrm{Cap}(x,t)$, $\mathrm{Cap}(x)$	capability vector	§2.4.2

$\mathrm{Lic}(x,t)$, $\mathrm{Lic}(x)$	license vector	§2.4.2
$\mathrm{Lic}^0(x,t)$, $\mathrm{Lic}^0(x)$	maximum file license	§2.3, §2.4.2
$\mathrm{Priv}(x,t)$, $\mathrm{Priv}(x)$	privilege vector	§2.4.2
$H(f)$	pex-holding process	§2.4.3
$APX(f)$	accept pex indicator	§2.4.3
$X(f)$	pexity indicator	§2.4.3
$AM(p)$	process audit mask	§2.4.4
SAM	system audit mask	§2.4.4
$PC(f)$	poison class	§2.4.4
$PM[i]$	poison mask	§2.4.4

2.2. Labels

A label can be any element of a given finite lattice $\mathbf{L}$, or the special symbol **y**, or the special symbol **n**. Let $\mathbf{L}^* = \mathbf{L} \cup \{\mathbf{y}, \mathbf{n}\}$ denote this set of possible labels. The lattice $\mathbf{L}$ is a design parameter. We have chosen the lattice of subsets of 480 elements, represented as vectors of 480 bits.

☞ The bit vectors 000 < 001 < 011 < 111 might represent the customary classification levels: unclassified, confidential, secret, top secret. Further bits might represent compartments: 000 100 for Iran, 000 010 for Nicaragua, 000 001 for submarine traffic, etc. Oliver North was cleared at least for 111 110.

Let the ordering relation on $\mathbf{L}$ be denoted $\leq$, the meet operation inf, the join operation sup, bottom element $\perp$ and top element ` . The only comparison predicates we use are =, $\leq$, and $\not\leq$. The predicate $\not\leq$ means ''not $\leq$'' or ''is not dominated by''; it should not be thought of as >.

A data flow $x \rightarrow y$ is said to be *up* if $L(x) \leq L(y)$. Otherwise the flow is *down*.

☞ Up describes a $\leq$ relation and down describes $\not\leq$. By always referring to the direction of flow, we avoid the common, but confusing, phrases ''write up'' and ''read down,'' both of which describe upward flow.

We extend the meaning of $\leq$ to $\mathbf{L}^*$: for all x in $\mathbf{L}^*$, $x \leq \mathbf{y}$, $\mathbf{y} \leq x$, $\sup(x, \mathbf{y}) = \inf(x, \mathbf{y}) = x$, and $\sup(x, \mathbf{n}) = \inf(x, \mathbf{n}) = \mathbf{n}$. For all x in $\mathbf{L}$, $x \not\leq \mathbf{n}$ and $\mathbf{n} \not\leq x$. Also $\mathbf{n} \not\leq \mathbf{n}$. Note that $(\mathbf{L}^*, \leq)$ is not a lattice, and that $\leq$ is only partially transitive on $\mathbf{L}^*$ in that $L_1 \leq L_2$ and $L_2 \leq L_3$ imply $L_1 \leq L_3$ only if $L_2 \in \mathbf{L}$.

☞ Label **y** (yes) is intended for files such as `/dev/null` that may always be read or written. A place labeled **y** is amnesiac; what comes out is unrelated to what goes in. Label **n** (no) is intended for files that may never be read or written except by trusted processes.

2.3. Privileges and superuser

File permissions behave as always, except that superuser status does not automatically confer write permission. Historically restricted operations such as mounting a file system or changing userid are still reserved to the superuser. However, most such operations require privilege in addition to superuser status.

There are two classes of privileges, called capabilities and licenses. Privileges are stored with labels; system calls that set and retrieve labels handle privileges at the same time.

Users may be given *licenses* to perform security-related tasks. Licenses may persist across *exec* and can be given up at any time. To exercise a privilege, a process with a license for that privilege must be running a program marked with a corresponding *capability*. Thus, in general, sensitive actions can only be performed by trusted users using trusted software.

Files, too, may have licenses. The licenses of an executable file augment the inherited licenses of processes that execute the file. File licensing is limited by a permanent maximum file license Lic^0; the effective set of licenses for file f is $\mathrm{Lic}(f) \wedge \mathrm{Lic}^0$.

☞ File licensing is much like the classic set-userid mechanism. Any user who can execute a licensed program automatically enjoys the privileges of the program. Unlike the effective userid, however, a license obtained from a file on *exec* is not inherited by child processes.

Capabilities of a process are computed from the licenses inherited from its parent (ultimately from the initialization process) and from the capabilities and licenses of the program it is executing; see §3.3. The capabilities express the powers the process can acutally exercise.

☞ For example, to set userid, a superuser process must have capability $\mathrm{Cap}_{\mathrm{uarea}}$ (§3.3); to initiate or change accounting, it must have capability $\mathrm{Cap}_{\mathrm{log}}$; and to mount a file system or make an external medium accessible, it must have capability $\mathrm{Cap}_{\mathrm{extern}}$.

2.4. Ingredients

The following notions are added to the usual UNIX model.

2.4.1. Labels and ceilings

Each participant x in data flow has a time-varying **L***-valued label, $L(x, t)$, written $L(x)$ when t doesn't matter. Labels of processes and seek pointers are restricted to **L**.

Each participant x in data flow has a time-varying **L***-valued ceiling, $C(x, t)$, sometimes written $C(x)$. By convention all files in file system FS share the same ceiling, called $C(FS)$, i.e. $C(f) = C(FS(f))$. By convention, any entity x that lacks a specific ceiling has $C(x) = \grave{}$.

Each file system type has a default ceiling value.

Each file or process has a time-varying *fixity* attribute, $F(f)$ or $F(p)$, which governs the mutability of label $L(f)$ or $L(p)$, and takes on values

loose: Label or fixity may change.
frozen: Label may not change; fixity may change.
rigid: Fixity may not change; label may change only with privilege.
constant: Neither label nor fixity may change.

New system calls, *setflab*, *fsetflab*, *getflab*, *fgetflab*, *setplab*, *getplab*, set and query labels and privileges of files and processes; see §3.1.1.

2.4.2. Capabilities and licenses

Each process or file has a set of time-varying boolean capabilities, further described in §3.3: $\mathrm{Cap}_{\mathrm{setpriv}}(x, t)$, $\mathrm{Cap}_{\mathrm{setlic}}(x, t)$, $\mathrm{Cap}_{\mathrm{nochk}}(x, t)$, $\mathrm{Cap}_{\mathrm{extern}}(x, t)$, $\mathrm{Cap}_{\mathrm{uarea}}(x, t)$, $\mathrm{Cap}_{\mathrm{log}}(x, t)$. The six predicates together constitute a bit vector $\mathrm{Cap}(x, t)$. Predicates may be written $\mathrm{Cap}_{\mathrm{nochk}}(x)$, $\mathrm{Cap}(x)$, and so on, when time is unimportant.

☞ Capabilities $\mathrm{Cap}(p)$ are rights of a process to override security policy. The capabilities of a process are limited by the capabilities $\mathrm{Cap}(f)$ of the currently executing file f and are not inherited.

Each process or file has a set of boolean licenses $Uk(x)$, subscripted like capabilities and together constituting a bit vector $\mathrm{Lic}(x)$.

The set of capabilities and licenses of a file or process are collectively known as privileges and are denoted $\mathrm{Priv}(f)$ or $\mathrm{Priv}(p)$. A *trusted* predicate, $T(x)$, is defined on files as the logical OR of all the privileges,

$$T(f) = \bigvee_k (\mathrm{Cap}_k(f) \vee \mathrm{Lic}_k(f)),$$

and on processes as the OR of the capabilities,

$$T(p) = \bigvee_k \mathrm{Cap}_k(p).$$

maximum licenses Ux^0 with collective vector Lic^0.

Each file system has a set of boolean *privilege masks* $\mathrm{Cap}_k(FS)$ and $\mathrm{Lic}_k(FS)$, subscripted like capabilities, with collective vectors $\mathrm{Cap}(FS)$, $\mathrm{Lic}(FS)$, and $\mathrm{Priv}(FS)$.

Each file system type has a default privilege mask.

2.4.3. Private paths

Each open inode f has a *pex state*, comprising a *holding process* $H(f)$, a boolean *accept pex* indicator $APX(f)$, and a *pexity* indicator $X(f)$, which can take on three values:

unpexed: $H(f)$ is irrelevant.
pexed: Process $H(f)$ has exclusive access to f.
unpexing: f is unusable by any process until becoming **unpexed**.

A new family of IO controls manipulates the pex state; see §3.7.2.

Each stream has a *stream identifier*, which is an arbitrary string. Stream identifiers are retrieved by an IO control and set by a privileged IO control; see §3.7.3.

☞ In v10 and IX streams comprise pipes, terminals, and communication ports; pipes may be mounted in the file system. The stream identifier conventionally holds security-related information such as the trustedness and authentication record of a terminal port.

2.4.4. Auditing

Each process p has a *audit mask*, $AM(p)$. A *system audit mask*, *SAM*, determines the base level of auditing. Each file has a *poison class*, $PC(f)$, which takes integer values in the range 0 through 3. For each poison class i there is a *poison mask*, $PM[i]$.

☞ Process audit masks control auditing coverage and are inherited across fork and exec. The poison class of a file determines a poison mask that augments the audit mask of each process that deals with the file; see §3.6.8.

Logging data is collected in a new kind of special file; see §3.4.8.

A new system call *syslog* controls auditing; see §3.6.8.

2.4.5. Miscellaneous

There are two new file modes:

`S_IAPPEND` forces all writing to occur at the end of the file and prevents truncation by *creat*.

`S_IBLIND` makes a directory unreadable and immune to label checking; see §3.4.7.

There are two new error return codes:

`ELAB` is returned for security label violations; see §3.2.6.

`EPRIV` is returned for lack of privilege; see §3.2.7.

A new signal, `SIGLAB`, detects changes in labels of open files; see §3.2.5.

The old signal `SIGPIPE` may be triggered in a new way; see §3.2.6.

Each file descriptor in each process has a *safe-to-write* bit, a *safe-to-read* bit, and an *exempt* bit; see §3.6.5.

A new system call, *unsafe*, queries and resets safe-to-read and safe-to-write bits; see §3.6.9.

A new system call, *nochk*, changes exempt bits; see §3.6.5 and §3.6.9.

Two old system calls, *seek* and *tell*, have been recalled to active duty; see §3.5.14 and §3.5.18.

2.5. Formal policy

In the policy statements below, variables t_0 and t_1 represent times such that $t_0 \leq t_1$. The entities among which flows occur are not designated; for a complete list of recognized flows, see Table §3.1.1.

Certain system calls *renew* entities; a renewed entity is actually or nominally bereft of memory of past contents. If an entity x is renewed in the interval t_0 to t_1, there is no persistent flow $x \rightarrow x$ across that interval. The following actions are recognized as renewals.

Seeking relative to beginning of file renews the seek pointer.

Setting a process ceiling renews the ceiling.

Setting a file label away from **n** renews the file.

☞ Files labeled **n** are unobservable in the absence of privilege. Thus reclassifying to **n** is harmless.

☞ *Exec* with no arguments could be considered to be a renewal, but our understanding that *exec* starts a new process makes that unnecessary.

2.5.1. Generic policy

The policy is to be observed by all untrusted processes. Trusted process may deviate from the policy only in ways recorded as ''exceptions.''

Upward flow. If a flow $x \rightarrow y$ occurs at time t, it must be upward.

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

Exception: either x or y is a process p and $\mathrm{Cap}_{\mathrm{nochk}}(p, t)$ is true.

Monotone labels. A persistent flow $x \rightarrow x$ must be upward unless $L(x, t_{1)} = \mathbf{n}$. so it is harmless to relabel a file **n**.

$$L(x, t_0) \leq L(x, t_1)$$

☞ A file labeled **n** is inaccessible to untrusted processes,

Impenetrable Ceilings. If a flow $x \rightarrow y$ occurs at time t, it must respect the ceiling of the causative process p, and any ceilings of the participating entities.

$$\sup(L(x, t), L(y, t)) \leq \inf(C(p, t), C(x, t), C(y, t))$$

Exception: either x or y is a process p and $\mathrm{Cap}_{\mathrm{nochk}}(p, t)$ is true.

Monotone ceilings. A ceiling can only decrease.

$$C(x, t_1) \leq C(x, t_0)$$

Exception: x is a process p and $\mathrm{Cap}_{\mathrm{setlic}}(p, t_0)$ is true.

Inherited ceilings. At the time t_0 of starting, the ceiling of a new process q is dominated by that of the process p that started it.

$$C(q, t_0) \leq C(p, t_0)$$

In the absence of privilege, the policy forbids a chain of flows $x_0 \rightarrow x_1 \rightarrow \cdots \rightarrow x_n$, between entities observed at times $t_0 \leq t_1 \leq \cdots \leq t_n$, where $L(x_0, t_0) \not\leq L(x_n, t_n)$. The label policy also forbids such a chain of flows when $L(x_n, t_0)$ is defined and $L(x_n, t_0) \not\leq C(x_n, t_0)$.

☞ Correct implementation depends on identifying all entities and flows and then assuring that each flow respects the policy.

2.5.2. Privilege policy

These rules do not address the initial setting of capabilities in a process; see §3.3.

Monotone capabilities. The capabilities of a process p can only decrease.

$$\mathrm{Cap}(p, t_1) \leq \mathrm{Cap}(p, t_0)$$

Monotone licenses. The licenses of a process p can only decrease.

$$\mathrm{Lic}(p, t_1) \leq \mathrm{Lic}(p, t_0)$$

Exception: $\mathrm{Cap}_{\mathrm{setlic}}(p, t_0)$ is true.

Inherited licenses. At the time t_0 of starting, the license of a new process q is dominated by that of the process p that started it.

$$\mathrm{Lic}(q, t_0) \leq \mathrm{Lic}(p, t_0)$$

Persistent file privilege. The privileges of a file cannot be changed.

$$\mathrm{Priv}(f, t_1) = \mathrm{Priv}(f, t_0)$$

Exception: process p with capability $\mathrm{Cap}_{\mathrm{setpriv}}(p, t)$ causes a change during $t_0 \leq t \leq T_1$.

Persistent trusted files. A trusted file f cannot be written into.

$$x \rightarrow f \Rightarrow \neg T(f).$$

3. Details

This section specifies security check calculations (§3.1), the effect of label changes and `SIGLAB` (§3.2), the privilege mechanism (§3.3), labels of special files (§3.4), new system calls (§3.6), and special security behavior of system calls (§3.5, §3.7).

3.1. Security checks

Standard security checks are made for system calls that refer to files or file descriptors. Each system call is first subjected to the security check calculation specified in table §3.1.1 below, as elaborated in sections §3.5, §3.6, and §3.7. Failed security checks return error `ELAB` (§3.2.6) or `EPRIV` (§3.2.7) unless otherwise specified.

Table §3.1.1 summarizes the data flows caused by each system call, lists the standard checks performed for each call, and indicates renewal possibilities. Some calls have special checks, which are described in sections referred to in the Notes column. The standard checks are

READ(*d*)	for a *read* call with descriptor *d* (§3.1.3)
WRITE(*d*)	for a *write* call with descriptor *d* (§3.1.4)
R(*x*)	fre retrieving other data from an object *x*, as in *stat* (§3.1.5)
RS(*d*)	for retrieving the seek pointer of file descriptor *d* (§3.1.6)
W(*x*)	for assigning other data to an object *x* (§3.1.7)
WS(*d*)	for assigning to the seek pointer of file descriptor *d* (§3.1.8)
RD(*f*)	for interpreting a file name *f* (§3.1.9)
WRD(*f*)	for writing a file name *f* in a directory (§3.1.10)
P(*f*)	process-exclusive access (§3.1.11)
$\mathrm{Cap}_{\mathrm{log}}$	$\mathrm{Cap}_{\mathrm{log}}(p)$ must be true
$\mathrm{Cap}_{\mathrm{extern}}$	$\mathrm{Cap}_{\mathrm{extern}}(p)$ must be true
$\mathrm{Cap}_{\mathrm{uarea}}$	$\mathrm{Cap}_{\mathrm{uarea}}(p)$ must be true if superuser status is required

Information (usually concerning access rights) obtained only through error returns is not normally subject to security checks. Information (such as link counts) created as side effects of system calls is checked unless checking would seriously impair utility.

Data flows marked * are not checked. Bracketed entries in the data flow column refer to covert flows that are checked, although they do not count as data flows in the strict sense of §2.1.

☞ For example, it may be possible to infer the value of a seek pointer from the bits delivered by a read, although the value is not delivered directly. Hence a flow $[s \rightarrow p]$ is attributed to the *read* system call.

Symbols *f*, *d*, and *s* in the body of the table denote the file, file descriptor, and seek pointer (if any) referred to by the arguments of the system call. Thus for a system call with a file descriptor argument *d*, the symbol *f* means $f(d)$ and *s* means $s(d)$. Subscripts distinguish multiple file arguments. Symbol *q* refers to another process, in particular the new process after *exec*; $u(p)$ and $u(q)$ denote the user areas of the respective processes; *u* is short for $u(p)$. The remaining codes used in the table are

i	information about inode inferable through error return
p	implicit write to `/proc` or to inode of `/proc/`*p*
u	implicit write in uarea, readable through `/proc/`*p*`; contrast with explicit write` $p \rightarrow u$
X	abolish

☞ Consider `chmod("/etc/passwd",0666)`. It is necessary to read directory `/etc` to find the file `passwd` (RD). It is also necessary to read the inode of `/etc/passwd` to determine whether the process has the right to change it. This is an implicit read (i). If permission is granted, the given mode is written into the inode (W(*f*)). Finally the inode change date is set as a side effect.

☞ To do `unlink("/etc/passwd")`, it is necessary to read directory `/etc` to find the file `passwd` (RD, implied by WRD) and to read the inode of `/etc/passwd` to determine whether the process has the right to change it (i). If the link count does not go to zero, the new count must be written and the inode change time updated (W). Finally the entry for `passwd` must be deleted from directory `/etc` and the modification and change times for `/etc` must be updated (WRD).

The behavior of system calls with no marks in the table is unaffected. When both security and permissions are checked on a given file, security is checked first. An implementer may choose to return a search permission error encountered early in a path even if a security error would occur later in the path.

3.1.1. Table of system calls

System call	Priv	Data flows	Checks	Notes
access(f,m)			RD(f)	i
acct(f)	Cap_{log}		RD(f)	i §3.5.1
alarm(n)				
biasclock(n)				§3.5.17
brk(n)		$p \rightarrow u$		
chdir(f)		RD(f)	iu	
chmod(f,m)		$p \rightarrow f$	RD(f),W(f)	i §3.5.2
chown(f,n_1,n_2)		$p \rightarrow f$	RD(f),W(f)	
chroot				X
close(d)				u §3.5.4
creat(f,m)		$p \rightarrow f$	P(f),WRD(f)	iu §3.5.5
dirread(d)		$f \rightarrow p$ $[f \rightarrow s]$ $[s \rightarrow p]$	READ(d)	i §3.5.13
dup(d)				u §3.5.6
dup2(d_1,d_2)				u §3.5.6
exec(f,a)		$p \rightarrow q$ $f \rightarrow q$ $u(p) \rightarrow u(q)$*	RD(f)	iup §3.5.7
exec(f,0)		$f \rightarrow q$ $u(p) \rightarrow u(q)$*	RD(f)	iup §3.5.7
exit(v)				p §3.5.22
fchmod(d,m)		$p \rightarrow f$	W(f)	
fchown(d,n_1,n_2)		$p \rightarrow f$	W(f)	
fgetflab(d,l)		$f \rightarrow p$	R(f)	§3.6.2
fmount(n_1,d,f,n_2)	$\text{Cap}_{\text{extern}}$	$p \rightarrow f$	RD(f),W(f)	i §3.5.8
fmount5(n,d,f,n,c)	$\text{Cap}_{\text{extern}}$	$p -> f$	RD(f),W(f)	i §3.5.8
fork()		$u(p) \rightarrow u(q)$		up
fsetflab(d,l)		$p \rightarrow f$		§3.6.6
fstat(d,b)		$f \rightarrow p$		R(f)
ftime(b)				
funmount(f)		$p \rightarrow f$	RD(f),W(f)	
getegid()		$u \rightarrow p$		
geteuid()		$u \rightarrow p$		
getflab(f,l)		$f \rightarrow p$	RD(f),R(f)	§3.6.2
getgid()		$u \rightarrow p$		
getgroups(n,b)				X
getlogname(b)		$u \rightarrow p$		
getpgrp(p)		$u \rightarrow p$*		
getpid()		$u \rightarrow p$		

System call	Priv	Data flows	Checks	Notes
getplab(l,c)			RS($C(p)$)	§3.6.3
getppid()		$u \rightarrow p$		
getuid()		$u \rightarrow p$		
ioctl(d,n,b)		various	P(f),etc.	§3.7
kill(q,n)		[$p \rightarrow q$]		p §3.5.16
labmount(d,l)		$f \rightarrow p$		§3.6.4
link(f_1, f_2)		[$p \rightarrow f_1$]	RD(f_1),WRD(f_2),W(f_1)	
lock(n)				u
lseek(d,n_1,n_2)		$p \rightarrow s$	P(f),WS(d),RS(d)	§3.5.10
		$s \rightarrow p$		
lstat(f,b)		$f \rightarrow p$	RD(f),R(f)	
mkdir(f,m)		$p \rightarrow f$*	WRD(f)	§3.5.11
mknod(f,m,b)		$p \rightarrow f$*	WRD(f)	§3.5.11
nap(n)				
nice(n)				u §3.5.12
nochk(d,m)				§3.6.5
open(f,n)			RD(f)	iu
pause()				u
pipe()				u
profil(b,n_1,n_2,n_3)				
read(d,b,n)		$f \rightarrow p$	P(f),READ(d)	§3.5.13
		[$f \rightarrow s$]		
		[$s \rightarrow p$]		
	[$p \rightarrow s$]			
readlink(f,b,n)		$f \rightarrow p$	RD(f),R(f)	
reboot(n)				
rmdir(f)		WRD(f)		
seek(d,n_1,n_2)		$p \rightarrow s$	P(f),WS(d)	§3.5.14
select(n_1,b_1,b_2,n_2)				§3.5.15
setflab(f,l)		$p \rightarrow f$	RD(f)	i §3.6.6
setgid(n)	$\text{Cap}_{\text{uarea}}$	$p \rightarrow u$		
setgroups(n,b)			X	
setlogname(b)	$\text{Cap}_{\text{uarea}}$	$p \rightarrow u$		
setpgrp(p,0)			u §3.5.9	
setpgrp(p,n)	$\text{Cap}_{\text{uarea}}$	$p \rightarrow u$		§3.5.9
setplab(l,c)				u §3.6.7
setruid(n)	$\text{Cap}_{\text{uarea}}$	$p \rightarrow u$		up
setuid(n)	$\text{Cap}_{\text{uarea}}$	$p \rightarrow u$		up
signal(n,g)				u §3.5.16
stat(f,b)		$f \rightarrow p$	RD(f),R(f)	
stime(b)				§3.5.17
symlink(f_1, f_2)		$p \rightarrow f_2$	WRD(f),W(f_2)	i
sync()				
syslog(n_1,n_2,n_3)	Cap_{log}	various		§3.6.8
tell(d)		$s \rightarrow p$	RS(d)	§3.5.18
time(b)				
times(b)				§3.5.20
umask(m)		$p \rightarrow u$		
unlink(f)		[$p \rightarrow f$]	WRD(f)	i §3.5.21
unsafe(n,b_1,b_2)				§3.6.9

System call	Priv	Data flows	Checks	Notes
utime(f,b)		$p \rightarrow f$	RD(f),W(f)	
vadvise(n)				
vlimit(d,n_1,n_2)	$\text{Cap}_{\text{uarea}}$	$p \rightarrow u$		
vswapon(f)				§3.5.19
vtimes(b_1,b_2)	sys$\rightarrow p$			
wait(b)		$q \rightarrow p$		u §3.5.22
write(d,b,n)		$p \rightarrow f$ $[p \rightarrow s]$ $[s \rightarrow f]$	P(f),WRITE(d),W(f)	§3.5.23

3.1.2. Standard check computations

Security checks enforce *critical inequalities* among labels as required by the ''Data flow'' column of table §3.1.1. An inequality need not be checked if it is overridden by privilege, if it involves a seek pointer in a stream, or if its truth is implied by side knowledge. Such knowledge may be obtained from safe bits, or from the partial transitivity of ≤ and the invariants

$L(p) \leq C(p)$
$L(p) \in \mathbf{L}$
$C(p) \in \mathbf{L}$
$L(s) \in \mathbf{L}$, if the seek pointer is defined

3.1.3. READ(d): Check for the read system call

Let file descriptor d name (p, s, f). The critical inequalities are

$L(f) \leq C(f)$
$L(f) \leq L(s)$
$L(f) \leq L(p)$
$L(f) \leq C(p)$
$L(s) \leq L(p)$
$L(s) \leq C(p)$
$L(p) \leq L(s)$

☞ Seek pointer inequalities follow from the observation that reading entails direct flow $f \rightarrow p$ (the bits) and covert flows $[s \rightarrow p]$ (which bits), $[p \rightarrow s]$ (how many) and $[f \rightarrow s]$ (at end of file).

Do the following check as each block of data is copied to user space.

Let $M = \sup(L(p), L(s), L(f))$.

If d is marked safe-to-read the check succeeds.

Otherwise, if $\text{Cap}_{\text{nochk}}(p)$ and d is marked exempt the check succeeds.

Otherwise, if the critical inequalities hold the check succeeds.

Otherwise, if $L(f) \not\leq C(f)$ then **error**.

Otherwise, if $M \not\leq C(p)$ then **error**.

Otherwise, if $L(p) \neq M$ and $F(p) \neq$ **loose** then **error**.

Otherwise, establish the critical inequalities:

If $L(p) \neq M$ set $L(p) := M$ and propagate with CHP(p), §3.2.1.

If $L(s) \neq M$ set $L(s) := M$ and propagate with CHS(s), §3.2.3.

If no error occurred mark d safe-to-read.

3.1.4. WRITE(*d*): Check for the write system call

Let file descriptor *d* name (p, s, f). For streams, *s* doesn't matter; we suppose $L(s) = L(f)$. The critical inequalities are

$$L(p) \le C(f)$$
$$L(s) \le C(f)$$
$$L(f) \le C(p)$$
$$L(s) \le L(f)$$
$$L(s) \le C(p)$$
$$L(p) \le L(s)$$
$$L(p) \le L(f)$$

☞ Seek pointer inequalities follow from the observation that writing entails direct flows $p \to f$ (the bits) and covert flows $[s \to f]$ (which bits) and $[p \to s]$ (how many). Inequality $L(s) \le C(p)$ prevents *p* from interfering with a higher process through a shared seek pointer.

Do the following check as each block of data is copied out of user space.

Let $M = \sup(L(p), L(s), L(f))$ and $C = \inf(C(p), C(f))$.

If *d* is marked safe-to-write the check succeeds.

Otherwise, if $T(f)$ then **error**.

Otherwise, if $\mathrm{Cap}_{\mathrm{nochk}}(p)$ and *d* is marked exempt the check succeeds.

Otherwise, if the critical inequalities hold the check succeeds.

Otherwise, if $M \not\le C$ then **error**.

Otherwise, if *f* is the process file for process *q* and $M \not\le L(q)$ then **error**.

Otherwise, if $M \not\le L(f)$ and $F(f) \ne$ **loose** then **error**.

Otherwise, establish the critical inequalities:

If $L(p) \not\le L(s)$ set $L(s) := \sup(L(p), L(s))$ and propagate with CHS(*s*), §3.2.3.

If $L(f) \ne M$ set $L(f) := M$ and propagate with CHF(*f*), §3.2.2.

On error, raise signal `SIGPIPE`.

Otherwise, mark *d* safe-to-write.

☞ Some of the complexity of the check arises from the possibility that $L(f) =$ **y**.

☞ `SIGPIPE` has nothing to do with security. Just as with broken pipes, it stops processes when their output is unexpectedly being thrown away.

☞ *Write* calls may fail with `ELAB` or `ECONC` even though the corresponding *open* calls succeed; programmers should always take care to check for *write* errors explicitly.

3.1.5. R(*f*): Check for read-like calls on a file

The critical inequalities are

$$L(f) \le C(f)$$
$$L(f) \le L(p)$$
$$L(p) \le C(p)$$

If $\mathrm{Cap}_{\mathrm{nochk}}(p)$ the check succeeds.

Otherwise, if the critical inequalities hold the check succeeds.

Otherwise, if $L(f) \not\le \infty(C(f), C(p))$ then **error**.

Otherwise, if $F(p) \ne$ **loose** then **error**.

Otherwise, establish the critical inequalities:

Set $L(p) := \sup(L(p), L(f))$ and propagate with CHP(*p*), §3.2.1.

☞ The capability to omit checks overrides. If the process is not cleared to read the object, raise the process label.

3.1.6. RS(*x*): Check for other read-like calls

If x is a file descriptor d let s be $s(d)$.

Otherwise let s be $C(p)$.

The critical inequalities are

$$L(s) \le L(p)$$
$$L(p) \le C(p)$$

If $\mathrm{Cap}_{\mathrm{nochk}}(p)$ the check succeeds.

Otherwise, if d is marked safe-to-read the check succeeds.

Otherwise, if the critical inequalities hold the check succeeds.

Otherwise, if $L(s) \not\le C(p)$ or $F(p) \neq$ **loose** then **error**.

Otherwise, establish the critical inequalities:

Set $L(p) := \sup(L(p), L(s))$ and propagate with CHP(p), §3.2.1.

☞ This check is used only by *lseek*, §3.5.10, *tell*, §3.5.18, and *getplab*, §3.6.3. For *getplab* the label $L(s)$ is not the ceiling label $C(p)$, but is instead the label of the ceiling, $L(C(p))$.

3.1.7. W(*f*): Check for write-like calls on a file

The critical inequalities are

$$L(p) \le C(f)$$
$$L(f) \le C(p)$$
$$L(p) \le L(f)$$

If $T(f)$ then **error**.

Otherwise, if $\mathrm{Cap}_{\mathrm{nochk}}(p)$ the check succeeds.

Otherwise, if the critical inequalities hold the check succeeds.

Otherwise, if f is the process file for process q and $L(p) \not\le L(q)$ then **error**.

Otherwise, if $L(f) =$ **n** then **error**.

Otherwise, if $F(x) \neq$ **loose** then **error**.

Otherwise establish the critical inequalities:

Set $L(f) := \sup(L(p), L(f))$ and propagate with CHF(f), §3.2.2.

☞ Do not alter trusted files (except with a privileged *setflab*, §3.6.6). If the file is not cleared to receive from the process, attempt to raise the file label.

3.1.8. WS(*d*): Check for write-like calls on a seek pointer

Let $s = s(d)$.

The critical inequalities are

$$L(s) \le C(p)$$
$$L(p) \le L(s)$$

If d is marked safe-to-read the check succeeds.

Otherwise, if $\mathrm{Cap}_{\mathrm{nochk}}(p)$ and d is marked exempt the check succeeds.

Otherwise, if the critical inequalities hold the check succeeds.

Otherwise, if the real value of $L(s) = \sup(L(p), L(s))$ the check succeeds. (An artificial value of $L(s)$ may have been considered, §3.5.14.)

Otherwise establish the critical inequalities:

Set $L(s) := \sup(L(p), L(s))$ and propagate with CHS(*s*), §3.2.3.

☞ This check is used only by *lseek*, §3.5.10, and *seek*, §3.5.14.

3.1.9. RD: Interpret a file name

Let directories f_i, $i = 1, 2, \cdots$, be visited in tracing the pathname. (The current directory is visited in tracing a relative pathname.)

For each f_i check R(f_i).

If the check fails then **error**.

Otherwise, if $L(f_i) \not\leq C(f_i)$ then **error**.

☞ Directory search does not involve data flow, so the label check is something of a frill. However, it stops a potential 150bps covert channel and prevents processes from peeking above their ceilings. (Bit rates are explained at §3.5.)

☞ The labels of symbolic links, like their permissions, are not checked; the labels of the substituted pathname suffice.

3.1.10. WRD: Write in a directory

Directories are written whenever entries are made or deleted.

Perform the RD check. Unless the directory is blind, §3.4.7, perform the W check for the directory as if it were a plain file being written.

☞ Although deleted entries are discernible only by processes authorized to read a directory, it would be prudent to clear them.

3.1.11. P(*f*): Process-exclusive check

If $X(f) \neq$ **unpexed** and $p \neq H(f)$ then error `ECONC`.

If *f* is one end of a pipe and $p = H(f)$

If $X(f) =$ **pexed** and $X(f') \neq$ **pexed** then error `ECONC`.

If $X(f) =$ **unpexing**, then error `ECONC`.

3.1.12. Atomicity

No label changes shall occur between making a security check and performing the action that the check is intended to protect. It is permissible, however, for label changes to intervene between checking a directory in a pathname and checking an entry in the directory. It may be necessary to check several times during an action, for example during a *read* that incurs several disk transfers or page waits.

3.2. Label changes

The degree to which a label is changeable is determined by its fixity, which takes on one of four values **loose**, **frozen**, **rigid**, and **constant**. The owner of a file with a loose or frozen label may change the fixity of that label to any value except **constant**. A process may change the fixity of its label back and forth between loose and frozen. There is no other way to change fixity.

Loose labels can be changed by any process, either as a result of an explicit label-changing system call or as a side effect of a security check calculation.

Frozen labels cannot be changed without first making the label loose.

Rigid labels can be changed only by processes with capability $\mathrm{Cap}_{\mathrm{extern}}$. The labels of external media are forced to be rigid.

Constant labels never change. Label constancy is a property of certain device files; see §3.4.

☞ External media have rigid labels because the label is the only record of what the external destination is allowed to see or of how incoming data should be classified.

Changes in labels are propagated by the following procedures. The actions described are to be taken immediately and atomically, even in the middle of a *read* or *write* call.

3.2.1. CHP(*p*): Propagate change in process label

Clear the safe-to-write bits on all file descriptors in process *p*.

If *p* has an associated process file *f*, propagate with CHF(*f*), avoiding further recursion.

☞ The inequalities $L(p) \leq L(f)$ are no longer known to be true for files *f* open in *p*. When *p* next attempts a write or when *p* resumes an incomplete write WRITE will be checked afresh.

3.2.2. CHF(*f*): Propagate change in file label

For all file descriptors *d* such that $f(d) = f$

Clear the safe-to-read and safe-to-write bits on *d*.

Raise signal `SIGLAB` in $p(d)$.

If *f* is the process file for process *q* then propagate with CHP(*q*), avoiding further recursion.

If *f* is a pipe end with other end f' then propagate with CHF(f'), avoiding further recursion.

☞ The inequalities $L(f) \leq L(q)$ are no longer known to be true in other processes *q*. When *q* next attempts a read or when *q* resumes an incomplete read READ will be checked afresh. Similarly the inequalities $L(f) \leq C(q)$ will be checked on future writes.

3.2.3. CHS(*s*): Propagate change in seek pointer label

Clear the safe-to-read and safe-to-write bits on all file descriptors *d* such that $s(d) = s$.

Raise signal `SIGLAB` in $p(d)$ for all such *d*.

3.2.4. New file descriptors

Perform the following operations on every new file descriptor and on every descriptor copied between processes, as by *exec* or `FIORCVFD`.

Clear the safe-to-read bit and safe-to-write bit.

Set the exempt bit.

☞ These rules are conservative. An implementer may copy the safe bits on descriptors cloned by *fork* or *dup* or by *open*ing a file descriptor file (`/dev/fd/*`). The values of exempt bits do not matter unless $\mathrm{Cap}_{\mathrm{nochk}}(p)$ is true.

3.2.5. SIGLAB

Signal `SIGLAB` is raised whenever a file descriptor changes label. `SIGLAB` is ignored if not caught.

☞ A trusted process with $\mathrm{Cap}_{\mathrm{nochk}}$ capability would use `SIGLAB` to prevent unintended downgrading that could occur if the labels of its input files changed. The process would most likely freeze its own label by setting $F(p)$ with *setplab* (§3.6.7), catch `SIGLAB`, and use *unsafe* (§3.6.9) to isolate changes when they occur.

3.2.6. ELAB

Attempts to violate critical label inequalities return error number `ELAB`.

☞ Information may be communicated through `ELAB`: Let process Low not be cleared for information known to process High, i.e. $L(\mathrm{High}) \not\leq L(\mathrm{Low})$, and let *f* be a file that Low is cleared for. High either does or does not contaminate (raise the label of) *f* by writing in it. Low tries to read *f*, and discovers which action High took according as it does or does not get error `ELAB`. The bandwidth of

about 80bps might be reduced by inserting delay when returning ELAB.

☞ Denning describes a similar channel, slower (10bps) but nonetheless elegant:[5] High either does or does not contaminate *f*. Low writes a 1 in another low file *g*, then reads *f* and from the contents determines whether *f* (and Low itself) has been contaminated. If *f* is not contaminated Low replaces the 1 by a 0. The still low file *g* now tells whether High did (1) or did not (0) contaminate *f*, yet in neither case does there exist a forbidden chain of data flow (in the sense of §2.5) from High to *g*, nor did any system call fail.

☞ Denied writes on a file descriptor raise SIGPIPE; see §3.1.4. It may also be desirable to stop processes that attempt too many security violations with a new, uncatchable signal, say SIGSPY.

3.2.7. EPRIV

Attempts to violate privilege rules return error number EPRIV.

☞ EPRIV affords covert channels as does ELAB. But processes that can modulate privilege must themselves be privileged; they have far more potent ways to make covert channels.

3.3. Privileges

The privilege bits $\mathrm{Priv}(p)$ and $\mathrm{Priv}(f)$ (recall that $\mathrm{Priv}(.)$ comprises both capabilities, $\mathrm{Cap}(.)$, and licenses, $\mathrm{Lic}(.)$) are stored with $L(p)$ and $L(f)$. The system calls *getplap, setplab, getflab,* and *setflab* retrieve and change privileges according to the policy set forth in §2.5.2.

Privileges $\mathrm{Priv}(p)$ are inherited across *fork*. Licenses $\mathrm{Lic}(p)$ may be inherited across *exec*; see §3.5.7. The initial capabilities of a child process *q* executing a trusted file *f* from file system $FS(f)$ are given by

$$\mathrm{Cap}(q) = \mathrm{Cap}(f) \wedge \mathrm{Cap}FS(f) \wedge (\mathrm{Lic}(p) \vee \mathrm{Lic}(f) \wedge \mathrm{Lic}(FS(f)) \wedge \mathrm{Lic}^0)$$

The compile-time parameter Lic^0 limits the self-licensing of files. Currently only $\mathrm{Cap}_{\mathrm{nochk}}$, $\mathrm{Cap}_{\mathrm{extern}}$, and $\mathrm{Cap}_{\mathrm{uarea}}$ may be self-licensed.

☞ The capabilities and licenses of a file are masked by the capability and license of its file system. A process obtains the capabilities it is licensed for from the capabilities of its executable file. Capabilities are licensed either by inherited licenses $\mathrm{Lic}(p)$ or by file licenses $\mathrm{Lic}(f)$.

☞ The utility of the self-licensing limit Lic^0 is questionable.

3.3.1. $\mathrm{Cap}_{\mathbf{extern}}$

This capability is used to introduce foreign data into the system. It is required for the *fmount* system call, to change labels away from **n**, and to change rigid labels.

☞ Capability $\mathrm{Cap}_{\mathrm{extern}}$ is necessary to open external media or to modify the label of an already open external medium. In general a $\mathrm{Cap}_{\mathrm{extern}}$ process will perform some authentication protocol to determine the proper label for the medium. Automatic label setting won't work because the system does not know what or who is out there.

☞ Label **n** may be used to hide data from (almost) everybody. Once marked **n**, it can't be read by normal means until resurrected by $\mathrm{Cap}_{\mathrm{extern}}$. In effect data marked **n** has been removed from the system; it comes back with a newly minted label as determined by an administrative program with capability $\mathrm{Cap}_{\mathrm{extern}}$.

3.3.2. $\mathrm{Cap}_{\mathbf{uarea}}$

This capability permits modifying ''user-area'' data that is readable by a descendent processes. In general, only the superuser can write such data (e.g. *setuid* and *setlogname*). Thus $\mathrm{Cap}_{\mathrm{uarea}}$ enforces the notion that the superuser has no business executing untrusted code.

☞ Non-superuser writing in the user area is controlled differently. *Umask* is censored when process labels drop; see§3.5.7. The process ceiling has its own label; see §3.6.7. The notion of abolishing $\mathrm{Cap}_{\mathrm{uarea}}$ and instead labeling each user-area item is appealing. However, we feared that this could lead to the necessity for some other privilege to undo label creep in the user area.

3.3.3. $\mathrm{Cap}_{\mathbf{nochk}}$

This privilege allows a process to ignore label comparisons. It is required for programs that inherently deal with multilevel data, for example programs to repair or back up file systems, or programs to handle multilevel multiplexed communication.

Capability $\mathrm{Cap}_{\mathrm{nochk}}$ overrides label checking only on file descriptors marked exempt. Fresh file descriptors are exempt; their exempt status may be changed by the *nochk* system call; see §3.6.5.

☞ $\mathrm{Cap}_{\mathrm{nochk}}$ and $\mathrm{Cap}_{\mathrm{extern}}$ may both be used to grant access to external media. The difference is that $\mathrm{Cap}_{\mathrm{nochk}}$ gives access only to the process that has the privilege—what is done with the access is under its control—while $\mathrm{Cap}_{\mathrm{extern}}$ makes the data available to other processes.

3.3.4. $\mathrm{Cap}_{\mathbf{setpriv}}$

This capability is required to change file privileges.

☞ Programs with capability $\mathrm{Cap}_{\mathrm{setpriv}}$ have the keys to the kingdom; they must be very carefully designed.

3.3.5. $\mathrm{Cap}_{\mathbf{setlic}}$

This capability is required to increase process licenses or to change the process label and ceiling arbitrarily.

☞ Capability $\mathrm{Cap}_{\mathrm{setlic}}$ is used to set up user sessions.

3.3.6. $\mathrm{Cap}_{\mathrm{log}}$

This capability is required to set up or change mandatory auditing.

3.4. Special files

Because special files address resources with unusual properties they may require unusual security considerations. Some files, for example `/dev/stdin`, have the property that the file descriptor obtained by *open* refers to a different object than the file name does. On such a file *fstat* and *stat* may return completely differing data; similarly *fchmod* may not affect the original file that was opened to obtain the file descriptor. Some special files have rigid or constant labels; it is impossible to change the fixity of these files.

3.4.1. Character special files.

On directories and character special files the accept pex indicator $APX(f)$ = **false** by default, otherwise $APX(f)$ = **true** by default. The setting of $APX(f)$ can be changed only for character special files (§3.7.2.4).

☞ Process exclusive (pex) requests are designed to secure trusted paths that are free from eavesdropping or from forging or corruption of data by other processes (§3.7.2). A typical application is demanding a password from a terminal. If, however, the ‘‘terminal’’ is really a communication line to another computer that is relaying the conversation, process-exclusive access is not enough to guarantee the privacy of the password. Only after a trusted process has somehow authenticated the trustedness of the external path can exclusive use of the internal path assure privacy (§3.7.2.3). Initially, then, devices must refuse to accept pex requests.

3.4.2. External media

External media comprise all special files not otherwise discussed in section §3.4. They are files over whose contents the system has not maintained complete control. Examples are terminals, communication links, tapes, and disks. Pipes are not external media.

When an external medium is not open, its label is **n**. The label is rigid and remains **n** after opening until it is set away from **n** by a trusted process; see §3.6.6. Further openings see this new label. The label reverts to **n** when no process has the file open.

☞ In general authentication is required before access may be granted to an external medium. Authentication will be administered by trusted processes.

☞ The fact that an inode shares a label with a file has the unfortunate effect that ordinary programs cannot read the properties of most device files.

3.4.3. Streams

When a stream is created, its stream identifier is initialized to the null string.

Both ends of a pipe stream share a single label, which is initialized to $\perp$.

A device stream shares its label with the device it is attached to.

3.4.4. Null and mem

The null device `/dev/null` has constant label **y**. The *stat* system call returns dummy data for `/dev/null`.

☞ This plugs covert channels through the mode bits of `/dev/null`.

The memory devices, `/dev/mem` and `/dev/kmem`, have constant label **n**.

3.4.5. Process files

For each file f in `/proc` $\mathrm{Priv}(f) = \mathrm{Priv}(p)$, where p is the corresponding process. If $\mathrm{Cap}_{\mathrm{nochk}}(p)$ was ever true, $L(f) =$ ` , otherwise $L(f) = L(p)$. The process file disappears with the process, regardless of whether it was trusted.

The virtual directory `/proc` has a **rigid** bottom label and has universal read permission. Creating a process does not count as writing in `/proc`.

☞ The top label for $\mathrm{Cap}_{\mathrm{nochk}}$ processes prevents leaks through debuggers.

☞ By modulating the rate of process creation, unrelated processes can communicate covertly through `/proc` at a rate of a few bits per second.

☞ When $\mathrm{Cap}_{\mathrm{nochk}}(p)$ is true upon *exec*, our implementation divorces the file privileges as well as the file label from that of the process. The divorce happens when the inode is created, which occurs on demand, not at process creation. Subsequent changes in process privilege are not reflected in the file. Thus, the privileges of a process file labeled ` may not agree with those of the process. But unless the file has capability $\mathrm{Cap}_{\mathrm{setlic}}$, its privileges dominate those of the process.

3.4.6. File descriptor files

The files `/dev/fd/*`, `/dev/tty`, `/dev/stdin`, etc, when referred to by file descriptor, share labels with the file descriptors they correspond to. When referred to by name, these files have the constant label **y**.

3.4.7. Blind directories

A new file mode `S_IBLIND` designates a directory as ‘‘blind.’’ No process can read a blind directory. Only the owner of a file can remove it from a blind directory. Only trusted processes can change blind mode.

☞ Blind mode is special pleading to preserve the semantics of `/tmp` – a compromise for compatibility. It affords an 80bps covert channel: High creates a file from some prearranged alphabet of names and Low tests whether the names are there. (Bit rates are explained at §3.5.)

☞ Trusted processes that place temporaries of known name in `/tmp` (or anywhere else) must take care to prevent improper access by other processes. For example, an untrusted user of a trusted process that wrote and later reopened a file might replace the file in the interim.

3.4.8. Log files

Security audit records are written to special files, `/dev/log*`. Data so written are actually appended to an associated ''repository'' file nominated by the privileged *syslog* system call, §3.6.8. A log file can be written by any process regardless of label and can be read by no process. Identifying information is automatically attached to each record written, so data cannot be forged and histories can be reconstructed. Direct writes on a repository file are silently discarded as on `/dev/null`. Repository files are protected by normal access control.

A distinguished log file, `/dev/log00` receives mandatory audit records in addition to data volunteered by writes. The intensity of mandatory auditing is controlled in each process by an audit mask, which is inherited across *fork* and *exec*; see §3.6.8.

Every file f has a poison class $PC(f)$. The poison class is normally invisible; it can be set or interrogated only with capability $\mathrm{Cap}_{\log}$. At each system call that mentions file f in a pathname, the poison mask $PM[PC(f)]$ is OR-ed into the process audit mask $AM(p)$.

☞ Data written on log files escape the formal policy. Unlimited covert channels via writing on a log file and reading from its repository are possible. A prudent administrative countermeasure is to nominate as repositories only files labeled ` or **n**; external media make particularly safe repositories.

☞ Poison classes allow administrators to stipulate extra logging when particular files are touched. Thus sensitive activities can be watched carefully without incurring a flood of low-value audit data from routine activities.

3.5. Security behavior of old system calls

☞ Bit-per-second (bps) estimates of covert channel bandwidth given below pertain to the research Tenth Edition (v10) running on a DEC VAX-11/750. The bandwidth could be held to the same level on faster machines by inserting a delay of about 100ms when *exec* is invoked with no arguments.

☞ Many reasonable estimates can be made with only a few basic constants and measurements: the open file limit (128 in v10), the number of forks a process can do per second (about 10), the number of files a process can create or open per second (about 80), and the number of message round trips per second possible across a pair of pipes (about 80). Some covert channels presuppose a population of files readily identifiable from content. We take 1000 as a population estimate, on the premise that a much larger population would call attention to itself.

☞ In making measurements some care must be taken to achieve fast process-switching rates. For example, in v10 it often helps to insert *nap* calls instead of busy-waits. In general it is real time, not user and system time, that counts. Also, because of overlaps, time measurements x and y often combine as $\sup(x, y)$ rather than as $x + y$.

☞ Covert channels often need synchronization: Low tells High, ''I got it,'' and then High sends another message. The rules allow Low to pipe to High, which make this easy.

3.5.1. acct(f)

The writing of shell accounting records is immune to label checks.

☞ Only a trusted process can nominate the accounting file. It must assure that the file is appropriately protected, perhaps by label ` , by label **n**, or by write-only transmission off line. Otherwise we have a 1000bps channel: High renames an executable file and executes it; Low reads the name from the accounting file.

3.5.2. chmod(f, m), fchmod(d, m)

There are two new *modes*, append-only, `S_IAPPEND`, and blind `S_IBLIND`, the latter being useful only with directories.

If f (or $f(d)$) is a directory and if blind mode is changed and $\neg\mathrm{Cap}_{\mathrm{extern}}(p)$ then error `EPRIV`.

☞ If unprivileged processes could change blind mode, High could create files in a virgin blind directory. Later, Low could turn blind mode off and read the names.

3.5.3. chown(f, u, g), fchown(d, u, g)

If *f* has setuid or setgid permission (mode 04000 or 02000) and either the new userid or the new groupid differs from the old then error `ECONC`.

If userid of *p* is not superuser

If userid of *p* does not own *f* then **error** `ECONC`.

If the new userid is not the same as the old then **error** `ECONC`.

If the new groupid is neither the same as the old nor the same as the effective groupid of the process then **error** `ECONC`.

☞ These rules have little bearing on the security policy. However, by using *chmod* or *chown*, the superuser can circumvent discretionary denial of write permission. This gambit should be highly visible in an audit trail.

3.5.4. close(d)

If *d* refers to a stream perform `ioctl(d, FIONPX, 0)`, but do not wait; see §3.7.2.2.

If the file has been read and $L(p) \leq L(f)$, update the file access time.

☞ To avoid the elaboration of a safe-to-write-access-time bit in every file descriptor, access times are not continually updated. The access-time check reduces a covert channel: High reads a file; Low spots the access. The bandwidth is limited, by the rate at which files can be opened, to about 50bps; the check is a cheap frill.

☞ A narrow (<100bps) covert channel using *close*: High selectively closes pipes, which Low detects with `EPIPE`.

3.5.5. creat(f, m)

If *f* exists, perform only the RD part of the WRD (write directory) check.

If *f* is a log file (§3.4.8) then error `ECONC`.

Otherwise, if *f* is new

Set $L(f) := \bot$.

Set $\mathrm{Priv}(f) := 0$.

Set $F(f) :=$ **loose**.

Perform the $W(f)$ check.

Otherwise

If the mode of *f* includes `S_IAPPEND`, do not truncate *f*.

If the size of *f* is nonzero, check $W(f)$.

Set $L(s) := \bot$.

Clear the safe-to-read and safe-to-write bits for the new file descriptor.

☞ Notionally file labels and seek pointers begin at $\bot$. However, *creat* writes mode bits into a new file, so the label rises immediately to $L(p)$.

☞ The $W(f)$ check on non-empty files closes an 80bps covert channel between unrelated processes: Low writes in a file; High optionally truncates it; Low detects which. In honest use, the label would probably rise anyway since *creat* is almost always followed by *write*.

3.5.6. dup(*d*), dup2(*d*,*d*)

☞ Covert channel: High opens some files, uses *dup*2 to selectively create more file descriptors, forks, and *exec*s a low process. Low infers which file descriptors are in use by attempting to read from them, thus learning one bit of information from the presence or absence of each possible file descriptor. If High picks among files that have read permission, no read permission, or a high label, the four possible outcomes for each file descriptor yield about 250bps. A similar bandwidth can be achieved

by High picking among a vocabulary of files which Low distinguishes by reading inode number or permission bits with *fstat*.

3.5.7. exec(f, arg, env), umask(m)

This description pertains to all flavors of *exec*.

Let *p* be the executor process and let *q* be the new process.

If *arg* and *env* are empty and no file descriptors have numbers greater than 3

Set $L(q) := \bot$.

If $L(p) \neq \bot$ set *umask* := 022.

Otherwise set $L(q) := L(p)$.

Perform the R(*f*) check (§3.1.5) in *q*, disregarding $\mathrm{Cap}_{\mathrm{nochk}}(q)$ and $F(q)$.

Clear all safe-to-read and safe-to-write bits in *q*.

Set all exempt bits in *q*.

Set $F(q) :=$ **false**.

Set process licenses.

If $T(f)$ set $\mathrm{Lic}(q) := \mathrm{Lic}(p)$.

Otherwise, set $\mathrm{Lic}(q) := 0$.

Set $\mathrm{Cap}(q)$ per §3.3.

Set $C(q) := C(p)$.

Set $AM(q) := AM(p) \vee SAM$.

Set $L(C(q)) := L(C(p))$.

Check R(*f*) in process *q*.

☞ The pex state (§3.7.1.1) of open files persists across *exec*.

☞ It is understood that no data will pass across *exec* in registers. This undocumented channel appears in some versions of UNIX, including v10.

☞ Various covert channels arise from the ‘‘drop-on-exec’’ feature, which gives a bottom label to a process when there is no memory via arguments. These channels involve inferring the values of freely settable uarea information: open files (identifiable by inode number or content), current directory (by inode number), program text file (by content). To keep the bandwidth down, drop-on-exec pertains only to processes that have at most the four default file descriptors (standard input, standard output, standard error, and control stream) open.

☞ (1) Parent High opens three low files before *fork* and *exec*; child Low identifies them by *fstat* and writes the results in the fourth open file (230bps). This is the widest known covert channel.
(2) High executes a sequence of low files, which record the sequence (100bps).
(3) Parent High sets current directory; child Low records it (80bps).

☞ The permission mask, being directly writable and readable by *umask*, could provide a direct channel on drop-on-exec. That channel is closed by censoring the mask to a fixed value.

☞ Licenses are inheritable only by trusted code. If untrusted code could inherit licenses, it could do nothing bad directly, but it could pass the licenses to trusted code along with bogus arguments. Then all trusted code would have to contain defenses against the possibility.

3.5.8. fmount(n, d, f, m), fmount5(n, d, f, m, Cp)

Let *FS* be $FS(f(d))$.

If the system call is *fmount*, set $C(FS)$ and $\mathrm{Priv}(FS)$ to their default values. The default ceiling $C(FS)$ is ` on all file system types except network file systems, where it is $\bot$. The default privilege mask $\mathrm{Priv}(FS)$ is all zeros.

If the sysem call is *fmount5,* set $C(FS)$ and $\mathrm{Priv}(FS)$ from the values pointed to by *Cp*.

3.5.9. getpgrp(q), setpgrp(q, n)

If $q = 0$ set $q := p$.

Otherwise, if $q \neq p$ then **error**.

If the operation is *setpgrp* and $n \neq p$ and $\neg \mathrm{Cap}_{\mathrm{uarea}}$ then **error**.

☞ The call `setpgrp(p, p)` (or `setpgrp(0, p)`) is similar to the `TIOCSETPGRP` *ioctl* call; and is quite common in practice. The requirement for privilege to set *n* arbitrarily avoids untrusted data flow through the process group.

☞ The ability to set the process group on another process would necessitate special label checks. As that ability is not used in v10 software it was deemed not worth the trouble. (''Job control'' shells, which use the ability, have never caught on in v10 – partly because windows largely subsume job control.)

3.5.10. lseek(d, o, n)

This call is equivalent to, and checked like, *seek(d, o, n)* followed by *tell(d)*; see §3.5.14 and §3.5.18.

☞ Seek pointers must be protected to stop a wide channel: High sets a shared seek pointer; Low reads it (3000bps for one file, proportionately more with many shared pointers). If a seek pointer had the same label as its file it would be impossible to read strictly up because the requirements $L(f) \leq L(p) = L(s)$ (*read* changes the seek pointer) would degenerate to $L(f) = L(p)$. Hence seek pointers have separate labels.

3.5.11. mkdir(f, m), mknod(f, m, a)

Set $L(f) := \bot$, except where specified differently for special files, §3.4.

Set $\mathrm{Priv}(f) := 0$.

Set $F(f) =$ **loose**.

Perform the $W(f)$ check.

If the operation is *mknod,* set the groupid of *f* to be the same as the groupid of the containing directory; if this differs from the groupid of *p* delete setgid from the mode of *f*.

☞ Notionally file labels begin at $\bot$. However, mode bits are written into a new file, so the label rises immediately to $L(p)$.

☞ A blind directory (§3.4.7) cannot be created directly, because *mkdir* heeds only the 9 file permission bits in *m*.

3.5.12. nice(n)

☞ In some systems (not v10) *nice* returns a value and could be used as a covert channel, at least by the superuser.

3.5.13. read(d, b, n); dirread(d, b, n)

If *d* refers to a blind directory, then **error**.

3.5.14. seek(d, o, n)

This call is like *lseek,* but returns an integer, –1 for failure and 0 for success.

If $n \neq 1$ or $o \neq 0$ check $P(f(d))$, §3.1.11.

Let file descriptor *d* name (p, s, f).

If $n = 0$ check $\mathrm{WS}(d)$ (§3.1.8) as if $L(s)$ were equal to $\bot$.

If $n = 1$ check $\mathrm{WS}(d)$.

If $n = 2$ check $\mathrm{WS}(d)$ as if $L(s)$ were equal to $L(f)$.

☞ Do not disturb *d* if some process other than *p* has process-exclusive access to it. On seeking relative to the beginning, the previous state of *s* is forgotten, so its previous label is irrelevant. On seeking to the end, the new value of *s* depends on the size of *f*, but not on the old value of *s*.

☞ This function, resurrected from earlier UNIX systems, avoids unnecessary label inflation that could happen with *lseek*; see §3.5.10.

☞ An untrusted process that shares an open file with a trusted process may by moving the seek pointer be able to insert information into trusted writes. In this way a process could influence downward writes or writes above its ceiling. To be safe, trusted processes should assert exclusive access over possibly shared file descriptors; see §3.7.2.1.

3.5.15. select(n, rd, wd, t)

Delete any descriptor *d* such that $X(f(d)) = X(f'(d)) =$ **pexed** and $H(f(d)) \neq p$ from the sets *rd* and *wd*.

If $X(f'(d))$ for some file descriptor *d* changes while waiting in *select*, report *d* as ready; see §3.7.2.

☞ A file held in process-exclusive state by other processes is not ready in *p*. A file in an impure process-exclusive state may need attention.

☞ Covert channels: (1) High writes on one of several pipes; Low uses *select* to discover which; High empties the pipe by reading it. (2) Low fills several pipes; High reads from one; Low uses *select* to discover which.

3.5.16. signal(s, fp), kill(p, s)

Let *L* be the label of the signal source.

If a signal would be caught and $L \not\leq L(p)$ then the signal is ignored.

If a core image is required, it will be made as if by *creat* and *write.* However, if $\mathrm{Cap}_{\mathrm{nochk}}(p)$ was ever true, no core image will be made. The condition, ''$\mathrm{Cap}_{\mathrm{nochk}}(p)$ was ever true,'' is inherited across *fork* but not across *exec*.

☞ Downward signals provide a covert channel of only 100bps. Stopping them is technically difficult, so we have not done so in our experimental system.

☞ A process with capability *Tnocheck* is trusted, and hence can be counted on not to spill its secrets across *exec* even if it relinquishes trustedness.

3.5.17. stime(t), biasclock(m)

☞ By diddling the clock, a superuser can communicate to an unrelated process. The channel is highly exposed; superusers have better ways to cheat.

3.5.18. tell(d)

Return, as a long, the current value of the seek pointer.

☞ This call is resurrected from earlier UNIX systems; see §3.5.14.

3.5.19. vswapon(f)

Legitimate values of *f*, which are built into v10, must be confined to nonremovable media.

☞ No privilege has been required for this system call that sets the swap device, because suitable devices are automatically labeled **n**. Compile-time conventions have been relied on to prevent an untrusted superuser program from diverting swaps to a removable device.

3.5.20. times(b), vtimes(b)

☞ A fraction of a bit per second may be communicated from child to parent through *times*, around 10bps through *vtimes.*

3.5.21. unlink(f), rmdir(f)

If $T(f)$ then **error**.

If f is in a blind directory and userid of p is not the owner of f then **error**.

If $\neg \mathrm{Cap}_{\mathrm{nochk}}(p)$ and $L(f) \not\preceq C(p)$ then **error**.

☞ No process, not a even trusted process, may unlink a trusted file. To deter spoofing by file substitution in `/tmp`, only a file's owner may delete it from a blind directory. A process may not delete files that it can't see data in.

☞ Covert channel: Low creates a bunch of files and places links to them in the blind directory `/tmp`. High unlinks them selectively; Low detects the change in link count and replenishes the links.

☞ A *W* check on unlink would narrow the covert channel. But a *W* check would have a nasty side effect: innocently created files could get stuck so no combination of untrusted processes could remove them. For example, suppose Low creates, and freezes the label of, a file in a directory that High subsequently raises above Low's ceiling. Low cannot remove the file because it can't see the directory. High, or any other process that can see the directory, will fail a W check because the file's label is frozen. Only Low, the file's owner, can loosen the label. The directory can't be deleted because the file is in it. Both file and directory are stuck. (High might get help from the superuser in unfreezing the file, but there is no guarantee that an unprivileged superuser can see all files that High can.)

3.5.22. wait(b), exit(s)

Let q be the exiting process.

If either the exit or the termination code of q is nonzero and $L(q) \not\preceq L(p)$, the status reported by *wait* shows exit code 0 and termination code `SIGTERM`.

☞ The status is censored to prevent downward data flow; a 10bps covert channel remains.

3.5.23. write(d, b, n)

If the file mode of d includes `S_IAPPEND`, write at the end of file, regardless of the seek pointer. Increment the seek pointer by the number of bytes written.

If $f(d)$ is nominated as a log file (§3.6.8) then error *ECONC*.

3.6. New system calls.

3.6.1. fmount5(n, d, f, m, Cp)

See §3.5.8.

3.6.2. getflab(f, Lp), fgetflab(d, Lp)

These two system calls return the label on a file, specified either by file name or file descriptor. The label $L(f)$ and privileges $\mathrm{Priv}(f)$ are placed in the location *Lp* points to.

3.6.3. getplab(Lp, Cp)

Return the label, ceiling and privilege vector of the current process.

If pointer *Cp* is not zero

Check $\mathrm{RS}(C(p))$.

If the check succeeds, place $C(p)$ and a zero privilege vector in the location pointed to.

Otherwise place **n** in the location pointed to.

If pointer *Lp* is not zero, place $L(p)$ and $\mathrm{Priv}(p)$ in the location pointed to.

If the $\mathrm{RS}(C(p))$ check failed then **error**.

☞ The system calls *setplab* and *getplab* mediate data flow through the ceiling label. To enforce the

formal policy on this flow the ceiling label itself is labeled. In the absence of such enforcement, 5000bps could be passed downward on this channel.

3.6.4. labmount(d, Cp)

Return the ceiling of the file system in which file descriptor *d* resides.

If $f(d)$ is in a file system place $C(FS(f(d)))$ in the location *Cp* points to.

Otherwise place **y** in the location pointed to.

3.6.5. nochk(fd, code)

If *code* = 0 mark file descriptor *fd* not exempt and clear safe-to-read and safe-to-write bits in *fd*.

Otherwise, mark *fd* exempt.

Return 0 or 1 according as *fd* was not or was exempt before.

☞ Exempt bits are turned on by default (§3.5.7), although the opposite convention would be better. The present convention allows some administrative programs that need $\text{Cap}_{\text{nochk}}$ privilege to be identical with those on ordinary UNIX systems.

3.6.6. setflab(f, Lp), fsetflab(d, Lp)

These two system calls set the label on a file. The description applies to *setflab*; *fsetflab* is to *setflab* as *fchmod* is to *chmod.* The proposed new privilege vector Priv, fixity *F*, and label *L* are pointed to by *Lp*.

If userid of *p* is not superuser or owner of *f* then **error** `EPERM`.

If *f* is a process file then **error**.

☞ Prevent violations of the ceiling or increases in privilege of the process.

Check privilege:

If $\text{Cap}_{\text{setpriv}}(p)$ the check succeeds.

Otherwise, if $F \neq F(f)$ and userid of *p* is not the superuser or the same as the owner of *f*, then error `ECONC`.

Otherwise, if $T(f)$ then **error**.

Otherwise, if Priv is nonzero then **error**.

Otherwise, the check succeeds.

If the privilege check succeeds, check labels: the following label check:

If L = **y** then **error**.

Otherwise, if L = **n**

If $L(f) \leq C(p)$ the check succeeds.

Otherwise **error**.

Otherwise, if $L(f)$ = **n** and $\text{Cap}_{\text{extern}}(p)$ the check succeeds.

Otherwise, if $L(f) \not\leq L$ then **error**.

Otherwise, if $\text{Cap}_{\text{nochk}}(p)$ the check succeeds.

Otherwise, if $L(p) \leq L \leq C(p)$ the check succeeds.

Otherwise **error**.

If the label check succeeds, check fixity: the following fixity check:

If $F(f)$ = **constant** then **error**.

Otherwise, if F = **constant** then **error**.

Otherwise, if F = **rigid** and *f* is not a stream then **error**.

Otherwise, if $F(f)$ = **loose** the check succeeds.

Otherwise, if $F(f)$ = **frozen** and userid of *p* is the same as the owner of *f*, the check succeeds.

Otherwise, if $F(f) = \textbf{rigid}$ and $\mathrm{Cap}_{\mathrm{extern}}(p)$ the check succeeds.

Otherwise **error**.

If the fixity check succeeds, change the label:

If $F(f) = \textbf{rigid}$ then set $F := \textbf{rigid}$.

If $L(f) \neq L$, and if f is an external medium, then block all input/output activity for all openings of *f*, drain its output buffers, and flush its input buffers.

Set $F(f) := F$.

If $L(f) \neq L$ or $\mathrm{Priv}(f) \neq \mathrm{Priv}$, then set $L(f) := L$ and $\mathrm{Priv}(f) := \mathrm{Priv}$ and propagate with CHF(*f*); see §3.2.2.

Unblock input/output (if blocked).

☞ Processes can be marked trusted only with trusted tools. Trustedness also can be removed only by trusted tools, not to forestall security breaches, but to preserve the tools themselves.

☞ Labels can only go up. Trusted processes can upgrade anything. Trusted processes may downgrade only by copying. The labels of external are rigid; their labels can change only with capability $\mathrm{Cap}_{\mathrm{extern}}$; see §3.4.2.

☞ *Setflab* on a file labeled **n**, usually an external medium, renews the file (§2.5). A file may be downgraded by setting its label to **n** and then using capability $\mathrm{Cap}_{\mathrm{extern}}$ to set the label away from **n**.

3.6.7. setplab(Lp, Cp)

This system call sets the label, privilege, and ceiling of the current process. The proposed label *L*, privilege Priv, capability Cap, license Lic and fixity *F* are pointed to by *Lp*; the proposed ceiling by *Cp*. A zero pointer designates a proposed value equal to the current value.

Check privilege:

If Cap is not bitwise less than or equal to $\mathrm{Cap}(p)$ then **error**.

Otherwise, if Lic is bitwise less than or equal to $\mathrm{Lic}(p)$ the check succeeds.

Otherwise, if $\mathrm{Cap}_{\mathrm{setlic}}(p)$ the check succeeds.

Otherwise **error**.

If the privilege check succeeds, check labels:

If $L = \textbf{y}$ or if $L = \textbf{n}$ or if $C = \textbf{y}$ or if $C = \textbf{n}$ then **error**.

Otherwise, if $L \not\leq C$ then **error**.

Otherwise, if $L(p) \not\leq L$ and $\neg\mathrm{Cap}_{\mathrm{setlic}}(p)$ then **error**.

Otherwise, if $C \not\leq C(p)$ and $\neg\mathrm{Cap}_{\mathrm{setlic}}(p)$ then **error**.

Otherwise the check succeeds.

If the label check succeeds then

If $F = \textbf{rigid}$ or $F = \textbf{constant}$ then **error**.

Set $F(p) := F$.

If $L(p) \neq L$ set $L(p) := L$ and propagate with *CHP*( ); see §3.2.1.

If the process loses capability $\mathrm{Cap}_{\mathrm{nochk}}$ or $C(p) \not\leq C$ then clear all safe-to-read and safe-to-write bits in *p*.

Set $\mathrm{Priv}(p) := \mathrm{Priv}$.

Set $C(p) := C$.

If *Cp* is not zero set $L(C(p))$.

If $\neg\mathrm{Cap}_{\mathrm{setlic}}(p)$ set $L(C(p)) := L(p)$.

Otherwise, set $L(C(p)) := \bot$.

☞ The label of an untrusted process can only go up; the ceiling can only come down. Label and ceiling

may never cross. If the ceiling passes the label of open files, subsequent R(*f*) checks (§3.1.5) or W(*f*) checks (§3.1.7) will fail.

☞ *Setplab* does not observe the license Cap(f) of the file being executed or the maximum license Lic^0; these are used only for initializing after *exec.*

☞ The bits in the ceiling are labeled (by $L(C(p))$) because the ceiling is readable and to some extent writable by an untrusted process. If the ceiling were not labeled, a lowish process with an all-ones ceiling could leak more than 5000bps by twiddling the ceiling.

3.6.8. syslog(c, n, x)

Change or inquire about security auditing. Argument *n* is an integer, which may also be interpreted as a file descriptor, *d*, or as a process id, *q*.

Switch on *c* into

Case `LOGON`: nominate file $f(d)$ as the repository for the log file with minor device number *x*, §3.4.8.

Case `LOGOFF`: turn off logging on device *x*.

Case `LOGGET`: return poison mask $PM[n]$. $PM[4]$ means the system audit mask *SAM*.

Case `LOGSET`: set $PM[n] := x$. $PM[4]$ means *SAM*.

Case `LOGFGET`: return poison class $PC(f(d))$.

Case `LOGFSET`: set $PC(f(d)) := x$.

Case `LOGPGET`: return process audit mask $AM(q)$.

Case `LOGPSET`: set $AM(q) := x$.

Each bit of an audit mask designates a class of mandatory audit records. The classes are

`N`	uses of file names (calls to *namei* in the kernel)
`S`	seek calls
`U`	writes to the ‘‘uarea’’
`I`	accesses of inode contents: *stat*(2), *utime*(2), etc.
`D`	possession and use of file descriptors: *open*(2), *close*(2), *read*(2), *write*(2), etc.
`P`	process history: *exec*(2), *fork*(2), *kill*(2), *exit*(2)
`L`	explicit changes of labels: *setflab*(2), *setplab*(2)
`A`	all changes of labels
`X`	uses of privilege
`E`	`ELAB` error returns
`T`	uses of a traced file or process

The format of audit records varies with the kind of action recorded.

☞ Writing of audit records is immune to label checking; the only security check is that the process which sets `LOGON` is trusted and has been able to open file *d*. Whether *d* is open for reading or writing does not matter. Logging persists after file *d* is closed. Log files may share repositories.

☞ It is the duty of the trusted nominating process to assure that the repository is protected so that logging records cannot be read in violation of the security policy, §3.4.8.

3.6.9. unsafe(n, rp, wp)

This system call queries and selectively clears safe-to-read and safe-to-write bits.

Two bit strings, *rs* and *ws*, pointed to by *rp* and *wp* are indexed as in *select.* Let the actual safe-to-read bits and safe-to-write bits of all file descriptors constitute strings *rd* and *wd*. Only the first *n* bits of each string are considered.

Do simultaneously

Set $rs := rd$ and $ws := wd$.

If $\mathrm{Cap}_{\mathrm{nochk}}(p)$ then set $rd := rd \& \neg rs$ and $wd := wd \& \neg ws$.

☞ Covert channel: High has pipe to Low; High raises level; Low uses *unsafe* to discover it. The

channel runs dry when High hits its ceiling, so not many bits—probably less than 20—can be transmitted. The call might be restricted to trusted processes, which are expected to be its principal user; see §3.2.5.

3.7. Ioctl requests

3.7.1. Changed requests

3.7.1.1. ioctl(d, FIORCVFD, r)

An extra field added to the `passfd` structure pointed to by *r* returns the capabilities $T(q)$ of the sending process *q*.

3.7.1.2. ioctl(d, TIOCSPGRP, r)

If the third argument is not a null pointer (the stream is to be associated with the process group pointed to by *r*):

If userid of *p* is not superuser then error `EPERM`.

Otherwise, if $\neg \mathrm{Cap}_{\mathrm{uarea}}(p)$ then **error**.

☞ The process group of a stream, being readable and writable, is subject to the security policy. As the practical uses of this system call are akin to those of *setpgrp*, it is protected in the same way; see §3.5.9.

3.7.2. New requests for process-exclusive access

In these requests *f* is the file $f(d)$; f' is the other end when *f* is a pipe end; and *r* is a pointer, which, if nonzero, points to a structure that is filled in on non-error returns as follows:

```
struct  pexclude {
        int oldnear;   /* previous value of X(f) */
        int newnear;   /* new value of X(f) */
        int farpid;    /* 0 if f not pipe or X(f') = unpexed */
                       /* otherwise H(f') */
        int farcap;    /* if farpid ≠ 0, Cap(H(f')) */
        int faruid;    /* if farpid ≠ 0, userid of H(f') */
};
```

Process-exclusive requests applied to streams skip over line discipline modules; neither the requests nor their return values can be forged by using the message line discipline. The process-exclusive state is inherited across *exec*.

`FIOPX` and `FIONPX` affect the pexity of pipes as described by the following table. Changes in response to a request on file descriptor *d* occur at both ends; code pairs in the table represent $X(f)$ and $X(f')$.

old state	new state	
	`FIOPX`	`FIONPX`
00	10	00
01	11	01
02	ECONC	02
10	10	00
20	20	00
11	11	02
0 = **unpexed**, 1 = **pexed**, 2 = **unpexing**		

☞ The process-exclusive requests return *bona fides* of the far process for use in establishing trust. To help assure that such trust does not unwittingly persist beyond the duration of exclusivity, a pipe goes

into an **unpexing** state, state 2 in the table, when one end goes from **pexed** to **unpexed**. In this state the pipe is unusable and it remains so until both ends reach the **unpexed** state.

3.7.2.1. ioctl(d, FIOPX, r)

`FIOPX` attempts to obtain for *p* exclusive access to the file $f = f(d)$.

If $APX(f)$ = **false**

If *f* is a directory then error `EISDIR`.

Otherwise, error `EPERM`.

Flush the stream *f*.

Set $H(f) := p$.

If *f* is not a pipe set $X(f)$ = **pexed** and return 0.

Now *f* is a pipe.

Set $X(f)$ and $X(f')$ according to the table in §3.7.2.

If process $H(f')$ is waiting in

`FIOPX`, it awakens and returns 0.

`FIONPX`, it awakens and returns 1.

select, with f' among the enabled file descriptors, f' becomes ready.

read or *write* on f', it awakens and returns error `ECONC`.

If $X(f) = X(f')$ = **pexed** return 0.

Otherwise wait, with timeout, for an answering `FIOPX` or `FIONPX` at the other end.

If the wait times out, return 1.

3.7.2.2. ioctl(d, FIONPX, r)

This request is used to reject an exclusive-access request at the other end of a pipe or to terminate exclusive access on a stream $f = f(d)$.

Flush the stream *f*.

Set $H(f) := 0$.

If *f* is not a pipe set $X(f) :=$ **unpexed** and return 0.

Now *f* is a pipe.

Set $X(f)$ and $X(f')$ according to the table in §3.7.2.

If process $H(f')$ is waiting in

`FIOPX`, it awakens and returns 1.

`FIONPX`, it awakens and returns 0.

select with $d(f')$ among the enabled file descriptors, $d(f')$ becomes ready.

read or *write* on f', it awakens and returns `ECONC`.

If $X(f) = X(f')$ = **unpexed** return 0.

Otherwise wait, with timeout, for an answering `FIOPX` or `FIONPX` at the other end.

If the wait times out, return 1.

3.7.2.3. ioctl(d, FIOQX, r)

This request queries pex state without changing it.

Return 0.

3.7.2.4. ioctl(d, FIOAPX, r), ioctl(d, FIOANPX, r)

These requests specify whether a stream will accept process-exclusive access requests. The accept pex indicator $APX(f)$ is initialized automatically when stream $f = f(d)$ is first opened (§3.4.1), and remains constant until changed by `FIOAPX` or `FIOANPX` or until last close.

If f is a pipe or is not a stream then error `ENOTTY`.

If $\neg \mathrm{Cap}_{\mathrm{extern}}(p)$ then error `ECONC`.

If the request is `FIOAPX` set $APX(f) := \textbf{true}$.

If the request is `FIOANPX` set $APX(f) := \textbf{false}$.

3.7.3. New requests for stream identifiers

The requests `FIOGSRC` and `FIOSSRC` get and set stream identifiers, null-terminated strings of at most 32 characters. A stream identifier typically records security-related information about the stream.

3.8. ioctl(d, FIOGSRC, s)

Copy the stream identifier of *d* into the character array *s*.

3.9. ioctl(d, FIOSSRC, s)

If $\neg \mathrm{Cap}_{\mathrm{extern}}$ then error `ECONC`.

Copy the null-terminated string pointed to by *s* into the stream identifier of *d*.

3.9.1. Table of ioctl requests

Let *d* be the first argument of *ioctl.* In default of more careful analysis, requests should be checked with W($f(d)$); requests that return values should also be checked with R($f(d)$).

This table of particular requests is intended to be illustrative, not definitive. It covers only new requests and requests that are documented in the v10 manual, and omits all requests pertinent to networking. Being based on v10, it has little bearing on other versions of UNIX.

The action entries mean

(empty)	no security checks
R	check R($f(d)$), §3.1.5
W	check W($f(d)$), §3.1.7

Request	Man page	Action
`FIOCLEX`	*ioctl* (2)	
`FIONCLEX`	*ioctl* (2)	
`FIOACCEPT`	*connld*(4)	
`FIOREJECT`	*connld*(4)	
`MTIOCEEOT`	*mt* (4)	
`MTIOCGET`	*mt* (4)	R
`MTIOCIEOT`	*mt* (4)	
`MTIOCTOP`	*mt* (4)	W
`FIOANPX`	*pex* (4)	§3.7.2.4
`FIOAPX`	*pex* (4)	§3.7.2.4
`FIONPX`	*pex* (4)	§3.7.2.2
`FIOPX`	*pex* (4)	§3.7.2.1
`FIOQX`	*pex* (4)	§3.7.2.3
`PIOCGETPR`	*proc* (4)	R
`PIOCKILL`	*proc* (4)	like *kill*, §3.5.16
`PIOCNICE`	*proc* (4)	§3.5.12
`PIOCOPENT`	*proc* (4)	

Request	Man page	Action
PIOCREXEC	*proc* (4)	
PIOCRUN	*proc* (4)	
PIOCSEXEC	*proc* (4)	
PIOCSMASK	*proc* (4)	
PIOCSTOP	*proc* (4)	
PIOCWSTOP	*proc* (4)	
UIOCHAR	*ra* (4)	R
UIOREPL	*ra* (4)	W
UIORRCT	*ra* (4)	R
UIOWRCT	*ra* (4)	W
FIOGSRC	*stream*(4)	R, §3.8
FIOINSLD	*stream*(4)	W
FIOLOOKLD	*stream*(4)	R
FIONREAD	*stream*(4)	R
FIOPOPLD	*stream*(4)	W
FIOPUSHLD	*stream*(4)	W
FIORCVFD	*stream*(4)	R, §3.7.1.1
FIOSNDFD	*stream*(4)	W
FIOSSRC	*stream*(4)	W, §3.9
TIOCEXCL	*stream*(4)	
TIOCFLUSH	*stream*(4)	W
TIOCGPGRP	*stream*(4)	
TIOCNXCL	*stream*(4)	
TIOCSBRK	*stream*(4)	W
TIOCSPGRP	*stream*(4)	§3.7.1.2
TIOCGDEV	*tty* (4)	R
TIOCSDEV	*tty* (4)	W
TIOCGETC	*ttyld* (4)	R
TIOCGETP	*ttyld* (4)	R
TIOCSETC	*ttyld* (4)	W
TIOCSETP	*ttyld* (4)	W

References

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

[2] Flink, C. W. and Weiss, J. D., ''System V/MLS labeling and mandatory policy alternatives,'' *AT&T Tech. J.* **67**, pp. 53-64 ().

[3] Bendet, D., Ferrigno, J., Green, G. B., Hondo, M., Lund, E., and Salemi, C. A., ''Challenges of trust: enhanced security for UNIX System V,'' *Proceedings, Winter Uniforum Conference* (1989).

[4] Department of Defense Computer Security Center, *Department of Defense Trusted Computer System Evaluation Criteria*, US Department of Defense, Fort Meade, MD (15 August 1983).

[5] Denning, D. E. R., *Cryptography and Data Security*, Addison-Wesley, Reading, MA (1982).

A Tour of IX

Doug McIlroy

Jim Reeds

ABSTRACT

The IX experimental version of UNIX® supports dynamic security labels, integrity controls, and divided privileges. Examples of its use show how IX differs from classical systems, and give some hints about how cope with the differences.

Although this tour consists of simple examples, it is intended for UNIX experts. It touches on the actions of system administrators as well as of ordinary users.

Many of the examples show things that don't work, because that seemed like a quicker way to show what IX is really about and to give a feel for how to cope with its novelties than would a series of examples that always worked. The frustrations of these examples, which are not qualitatively different from the frustrations that a newcomer to UNIX may experience, should not be taken as characteristic of everyday use. By and large IX works just like any UNIXsystem. The differences only show up when you work in multiple security compartments or levels at the same time. Then the IX model is actually easier than that of other UNIX systems with labeled access control.

Logging in

The first thing you see when you attempt to log in is familiar.

```
login:
```

But after you answer with a login name, the password prompt is different. It may look like this,

```
Password(you:19818):
```

The prompt reminds you of who you claim to be, in case you didn't know. You may reply with a traditional password. The prompt also gives a 5-digit challenge string for a Secure Net Key (also known as Atalla) challenge box, which you may obtain from the system's security administrator. Unlock the box with its password, key the challenge string into the box, and type into the computer the first 5 characters of the response—all in lower case. If the box displays

```
9Ab34F70
```

type

```
9ab34
```

and you should be admitted. Every time you log in you get a different challenge.

Why all the challenge-box folderol? Because sometimes the system admits no alternative. IX is paranoid about eavesdropping. If you try to log in from some ‘‘untrusted’’ source, such as a modem in California or another computer, where unknown agents might be listening in, you *must* use the challenge box:

```
Password(TAPPED LINE:23740):
```

The line may not be tapped, but one never knows, so the system makes the pessimistic guess and reminds you not to use your ordinary password.

Once you're logged in, you can poke around as in an ordinary UNIX system. Try a few `ls`

commands:

```
$ ls .
$ ls /
$ ls -l /etc
```

You may see some surprises. Along with the listing of `/etc` appears

```
ls: /etc/pwfile: Security label violation
```

This is a file you're not cleared to see, and with good reason; it contains the challenge-box passwords. The file has a very high security *label*, or classification. It may seem odd that the label applies not only to the contents of the file, but also to its inode. The reason is that information, such as file mode and modification time, can be written in the inode. That information is, as far as the sytem can tell, as secret as anything else in the file.

Here is a classic security issue. Why would anybody put secrets in an inode? Of course no ordinary user would. IX, however, attempts to prevent dishonest as well as accidental disclosure of secrets. Any ordinary program could be a Trojan horse, attempting to slip secrets through the cracks. Inode information is just one of many possible cracks.

Labels

Every process and every file, including terminals, pipes, and even `/dev/null`, has a label. To find the label of your shell, type

```
$ getlab
proc lab                ------ ------    ffff 0000 0000 ...
proc ceil               ------ ------    ffff 0000 0000 ...
```

There is the process label, `ffff 0000 0000 ...`, a hex constant representing the first 48 of a total of 480 bits of label. (We apologize for the primitive representation. There ought to be symbolic names for common labels, but they haven't been implemented.) This particular label is the system *floor*, the standard unclassified starting place for all users. It is also the standard label for communicating with the unclassified outside world. The strings of minus signs have to do with the tricky matter of privilege. We'll come back to that later.

You can ask for the label of the terminal—actually for the labels of all open file descriptors (`-d`):

```
$ getlab -d
proc lab                ------ ------    ffff 0000 0000 ...
proc ceil               ------ ------    ffff 0000 0000 ...
fd 0                    ------ ------ R  ffff 0000 0000 ...
fd 1                    ------ ------ R  ffff 0000 0000 ...
fd 2                    ------ ------ R  ffff 0000 0000 ...
fd 3                    ------ ------ R  ffff 0000 0000 ...
```

For good measure, the process label has been reported again. All four default file descriptors, standard input, standard output, standard error, and control stream, have the same label—as they must, for they all refer to the same open file.

Besides a label, each process also has a *ceiling*, the highest label the process is allowed to deal with. When you first log in, you are not permitted to see anything higher than the floor. You may look below the floor, however. A process may look at any file with a bitwise lesser or equal label.

The all-purpose `getlab` command can also retrieve the label of a file.

```
$ getlab /etc/passwd
/etc/passwd             ------ ------    0000 0000 ...
```

The all-zero label, known as *bottom*, is visible from every process. Thus the classical password file is, as always, visible to everybody.

What's the point of having files with labels below the floor, which serves as the ''unclassified''

security level? Before we answer, we must discuss the basic rule for handling classified information.

> *The label of the destination of a data transfer must dominate (be bitwise greater than or equal to) the label of the source.*

In other words, data may only flow up. Unlike ordinary file permissions, which the owners of files can change at will, label restrictions are mandatory. The system enforces them automatically.

IX supports the usual file permission mechanism. When the permissions allow writing, an attempt to write high data into a low file can have one of two outcomes. The transfer may be prohibited, or the file label may change to cover the label of the source process. Depending on which outcome is preferred, files below the floor may be protected in two different ways. Their labels may be *frozen*, in which case writing down is prohibited, or their labels may be *loose*, in which case writing raises the label. It is impossible for data written by an ordinary process to have a label below the label of the process. Thus unauthorized tampering with a supposedly bottom file will, if it is possible at all, be exposed by the file's label changing away from bottom.

We think of the floor as the zero of labels, with labels below the floor being ''negative.'' Labels above the floor are concerned with protecting secrets. Labels below the floor are concerned with monitoring ''integrity'', i.e. with detecting unintended changes.

Let us look at some more labels.

```
$ getlab /bin /bin/cp
/bin                    ------ ------ F  0000 0000 ...
/bin/cp                 ------ ------    0000 0000 ...
```

Both files have bottom labels. One, the directory `/bin`, is flagged `F` for frozen. This is a prophylactic measure to prevent the directory's label from rising whenever somebody—mistakenly—creates a file in it from a high process. (Recall that file creation involves *writing* in the containing directory.)

Considerable trouble would ensue should the directory ever get a high label. All the files in it would be cut off from processes with lower ceilings. More subtly, the labels of all low processes that did have clearance to search the directory would automatically rise. Thus contaminated, the processes would spread the unwanted label like a disease. Mislabeled directories have justifiably been dubbed ''tar babies.''

The label of the program `/bin/cp` is not frozen, although it might well be. The penalty if it becomes wrong, however, is considerably less, because a program (except perhaps for a shell) is far less often consulted than is its directory. The program, of course, is still protected by permissions:

```
$ ls -l /bin/cp
-rwxrwxr-x  1  bin     bin    11264 Oct 16  1987 /bin/cp
```

`/bin/cp` can be written by user `bin` and by group `bin` only. In particular, the superuser cannot write in the file; in IX the superuser has been stripped of universal write permission. The superuser can still do damage by masquerading:

```
# /etc/su bin
$ cp trash /bin/cp
```

or by taking over the file

```
# /etc/chown root /bin/cp
# cp trash /bin/cp
```

Still, unless the superuser has managed to get the process label down to bottom (by use of privilege, which we shall discuss later) the incident will leave a tell-tale notice in the new label

```
$ getlab /bin/cp
/bin/cp                 ------ ------    ffff 0000 0000 ...
```

The floor label on a file that should be at bottom reveals that the file has been compromised. A system administrator installing new software could guard against handling it with a bogus `cp` by running with a ceiling below the floor; we'll see how later.

It's a good idea to have your home directory frozen at the login label. Work at other labels is best done in other directories. Just to check,

```
$ getlab $HOME
/usr/you                ------ ------ F  ffff 0000 0000 ...
```

Wisely, it is frozen. As one would expect, there is a `setlab` program to change things. A file's owner—and nobody else—may change a file from frozen to loose and back. Here we subtract (`-s`) the frozen indicator, which works.

```
$ setlab -s F $HOME
$ getlab $HOME
/usr/you                ------ ------    ffff 0000 0000 ...
```

Next we try to set the label lower by ''subtracting'' `ffff`. This, being a downward change in label, is illegal.

```
$ setlab -s ffff $HOME
/usr/you: Security label violation
```

Because it's wise to leave the home directory frozen, let's add (`-a`) the frozen indicator, `F`, back into the label.

```
$ setlab -a F $HOME
$ getlab $HOME
/usr/you                ------ ------ F  ffff 0000 0000 ...
```

Fixity, YES, and no

We have seen that a label may be ''frozen'' or ''loose.'' There are two more degrees of fixity, ''rigid'' and ''constant.'' Rigid labels cannot be changed without privilege. The file descriptors for a terminal are rigid, denoted `R` in a displayed label.

```
$ getlab -d
proc lab                ------ ------    ffff 0000 0000 ...
proc ceil               ------ ------    ffff 0000 0000 ...
fd 0                    ------ ------ R  ffff 0000 0000 ...
fd 1                    ------ ------ R  ffff 0000 0000 ...
fd 2                    ------ ------ R  ffff 0000 0000 ...
fd 3                    ------ ------ R  ffff 0000 0000 ...
```

The reason for the rigidity of a terminal's label is that data cannot be controlled after leaving the computer. Special negotiations were required to determine an acceptable label in the first place. The system cannot honor an arbitrary change in label, for only a single level of data transfer has been approved.

Constant labels, denoted `C` in a `getlab` display, are built in; they can never be changed. An example is `/dev/null`.

```
$ getlab /dev/null
/dev/null               ------ ------ CY 0000 0000 ...
```

It is also marked `Y` for the special label YES. A file so labeled can be read or written by processes of any label. This is justified for `/dev/null` because whatever goes in there never comes out.

Complementary to YES is the other special label **no**. A file so labeled can be read or written only with privilege. The usual use of label **no** is to prevent data from leaving the machine. Unopened external ports are labeled rigid **no**. Privileged programs, in particular `login`, can give a different label to the file. The label persists as long as any process has the device open, at which time it reverts automatically to **no**. Because label protection covers inodes as well as data, unprivileged processes cannot fetch the label from a **no** file. In the following example, the labels of two currently active login devices, `dk08` and `dk10`, can be seen; the **no** label of the currently unused device, `dk12`, cannot.

```
$ who
reeds    dk/dk08  May 14 05:59
you      dk/dk10  May 14 12:59
$ getlab /dev/dk/dk08 /dev/dk/dk10 /dev/dk/dk12
/dev/dk/dk08  [name]    ------ ------ R  ffff 0000 0000 ...
/dev/dk/dk10  [name]    ------ ------ R  ffff 0000 0000 ...
/dev/dk/dk12: Security label violation
```

The label of a device need not be the same as the label of an opening of the device. When they are different, `getlab` reports both. An example is `/dev/tty`.

```
$ getlab /dev/tty
/dev/tty      [name]    ------ ------ CY 0000 0000 ...
/dev/tty      [desc]    ------ ------ R  ffff 0000 0000 ...
```

The device is a constant YES; it can always be examined. The open file, though, is just another instance of file descriptor 3 and shares its label, which in this (normal) case has a rigid floor value.

Higher labels

When you register as a user, you are given clearance for data up to some maximum level. To work with data above the floor, but within your clearance, you need a higher-label session. Here's a try for a session with a top (all-1's) label. We apply to the ''privilege server'' to invoke a session-making command to do the trick.

```
$ priv session -l ffff...
Password(you:38510):
Sorry.
```

Too bad; not enough clearance.

```
$ priv session -l ffffa
Password(you:57747):
priv(session -l ffffa)? y
session -l ffffa (EXEC /bin/sh)? y
$ getlab
proc lab                ------ ------    ffff a000 0000 ...
proc ceil               ------ ------    ffff a000 0000 ...
```

Success. As a free service of *session*, the ceiling went up too. The fact that it went up at all proves that you are recorded as being cleared for at least that level.

Besides the password, two other questions were asked, one by `priv`, and one by `session`. This is more paranoia. Your request for a sensitive action was mediated by the shell. The shell, a horrendously complicated and highly spoofable program cannot be trusted to deliver the arguments to other programs as you typed them. Thus you were asked to confirm that what the trusted programs received is what you typed.*

There is nothing special about any of the bits beyond the floor group. The bits are often portioned out to different information ''compartments.'' You are evidently cleared for at least compartment

`0000 8000 0000 ...`,

which stands perhaps for payroll information, and `0000 2000 0000 ...`, perhaps labor relations. Ored together with the floor, these compartments make the full label `ffff a000 0000 ...`

Labels also can be used to indicate classical security levels ordered by increasing sensitivity. For example

* Later we'll see how you know that the whole dialog wasn't a spoof.

```
0000 0000 0000 ...unclassified
0000 0100 0000 ...confidential
0000 0300 0000 ...secret
0000 0700 0000 ...top secret
```

Notice that the values are counted 0, 1, 3, 7 rather than 0, 1, 2, 3 because labels are compared bitwise, not as numeric values.

Remembering that it's wise to do higher level work in a different directory, try making one.

```
$ mkdir classified
classified: Unknown error
```

Too bad, again. Things have been done in the wrong order. You intended to write the name of a new file into your home directory. The write is forbidden because that directory is frozen at a lower label than your current session. ''Unknown error'' should really be ''Security label violation,'' but not all standard programs yet know about this new error code.

Better leave the high session and start again.

```
$ <control-D>    (to leave high session)
$ mkdir classified
$ priv session -l ffffa
Password(you:57747):
priv(session -l ffffa)? y
session -l ffffa (EXEC /bin/sh)? y
$ getlab classified
classified: Security label violation
```

Now things are getting a bit ridiculous. Where are we?

```
$ pwd
/
```

This is crazy; and it is a lie. You are actually stuck in a black hole, a directory labeled **no**, which not even `pwd` can trace the path to. With perhaps an excess of zeal the privilege server, `priv`, starts each privileged process in the black hole and cleans out the environment. This prevents spoofing games with relative pathnames and environment variables. It also means that you have to do a bit of extra work and probably reexecute your profile to get a friendly shell in a differently labeled session.

```
$ cd $HOME
no home directory
```

Oops. The environment is empty.

```
$ cd /usr/you
$ getlab classified
classified              ------ ------    ffff 0000 ...
```

Now create a file in directory `classified` (which as yet is still labeled at bottom).

```
$ >classified/secretfile
$ getlab classified classified/secretfile
classified              ------ ------    ffff a000 0000 ...
classified/secretfile   ------ ------    ffff a000 0000 ...
```

The label of the directory `classified` rises to cover the label of the process that created `secretfile`. And `secretfile` bears the label of that process, too.

You may have guessed that the labels printed by `getlab` have blanks in them only for readability. The blanks are optional. We have already used additive and subtractive labels in `setlab`; absolute labels may be used as well. This is another way to freeze the label of the new directory:

```
$ setlab Fffffa classified
```

Besides preventing the label from rising automatically, freezing prevents anybody else—even the superuser—from tinkering with the label. As the file's owner, you can still change it with `setlab`. You can only change it upwards, however. Let's raise our ceiling and change the label up further.

```
$ priv session -C fffff # raise ceiling
$ setlab "ffff e" secretfile
```

Now let's write stuff into the file and try to read it back

```
$ echo hello >secretfile
$ cat secretfile
Terminated1
```

Trouble again. The ceiling is high enough, so `cat` can work, but the terminal is not labeled high enough and its label is rigid. Thus the output of `cat` is blocked. The process dies in the same way that it would from writing on a broken pipe.

If the ceiling were brought down to the session level, `cat` could not read the file. Try the `drop` command, which runs a process with the ceiling dropped to the current process label.

```
$ drop cat secretfile
```

Still silence. How come? It is all right for `secretfile` to be opened; no secrets are revealed thereby, for its name is known in a lower-labeled directory.* Trouble sets in only when the file is read. The silence of `cat` indicates that it is not careful about distinguishing read errors from end-of-file. Some commands are.

```
$ drop cp secretfile /dev/tty
cp: secretfile: Unknown error
```

''Unknown error'' really should be ''Security label violation.'' This little glitch simply shows that `cp`, like most code in IX, was taken over lock, stock, and barrel from v10, knows nothing about security labels.

Let's get rid of the file while we can

```
$ rm secretfile
```

A couple of EOTs will get us back to where we started.

```
$ <control-D>
$ <control-D>
$ getlab
proc lab                ------ ------    ffff 0000 0000 ...
proc ceil               ------ ------    ffff 0000 0000 ...
$ pwd
/usr/you
```

Now remove the classified directory.

```
$ rm classified
rm: classified: Security label violation
```

It didn't work. The directory is above the ceiling, where you're not supposed to meddle. You need a process label at floor and a ceiling above the file being removed. To get rid of it, we raise the ceiling for just one command. Remembering that the command is going to be executed from the black hole, we give a full pathname

```
$ priv session -C fffff -c rm $HOME/classified
```

* Actually a small secret, exactly one bit, is revealed, namely whether the file permissions permit opening. This covert channel is beneath our notice.

The usual verification dialog ensues.

With `mux` windows, the constant changing of sessions would not be necessary. A different window can be devoted to each kind of session you're likely to need. Of course you can't cut and paste from a high window to a low one, but otherwise the windows work normally. And if in a window you return from a high session to a previos lower one, the entire contents of the window gets wiped out with the comment, ''Sanitized window downgrade.''

Privilege

Sometimes the fundamental rule that data flows up the lattice of labels must be broken. Data must be declassified. Administrators must install new software labeled below the floor. External media, which are normally labeled NO, must be made accessible. File systems, which contain data of all labels, must be checked and repaired. And finally, the right to break the rules must be administered. All these actions are regulated by privileges.

Privilege is mediated by *licenses*, which go with processes, and *capabilities*, which go with files. The superuser has no *ex officio* licenses.

There is, for example, an ''extern'' privilege to make external media visible. A process with an extern license isn't enough, however. The license may be exercised only in ''trusted'' programs that have the corresponding capability. Thus the mount command, which brings file systems into view, has extern capability, indicated by the `x` in its label.

```
$ getlab /etc/mount
/etc/mount              --x--- ------    0000 0000 ...
```

To run the mount command successfully a process must have extern license—plus userid 0 for old times' sake.

Licenses are obtained from the privilege server `priv`, more colloquially called the ''priv server.'' The priv server verifies your rights and hands out exactly the privileges you need for exactly one command. To mount a file system, you might invoke

```
$ /etc/su
Password(root:74390):
# /etc/priv mount -r /dev/ra14 /mnt
Password(you:65330):
priv(mount -r /dev/ra14 /mnt)? y
```

First you become superuser, then you issue the mount command through the priv server. Consulting a database of command requirements and rights, the server finds that you are required to issue your password so that nobody can assume your privileges simply by finding your terminal unattended. Then, as protection against having received an improper request from a possibly dishonest shell, it prints the request it thinks you entered. Only after your confirmation is the request finally performed.

The form in which a privileged command is finally executed is controlled by the database. In the present example, a request for `mount` becomes an invocation of `/etc/mount`.

The full list of privileges is `guxnlp`, which appear in printed labels intermixed with minus signs in the same way that file permissions `rwxrwxrwx` appear in the output of `ls`. They mean

`g` Logging privilege. Only one program, `syslog`, has this capability.

`u` This relatively minor privilege allows changes to the ''uarea.'' It is required for system calls such as `setuid` and `setlogname`. It is privileged because uarea data is passed between processes without label checks. Were it not for this privilege, untrusted code could use the uarea to pass information from high to low processes.

`x` Extern privilege allows new data sources to become accessible. It is needed to mount file systems, to give labels to terminals, or to downgrade (declassify) labels.

`n` Nocheck privilege bypasses label comparisons. It makes any data source available to the privileged program, which is presumed will treat the data with due respect. Nocheck privilege is weaker than

extern privilege, which makes data accessible to untrusted processes as well.

`l` Setlic privilege (a slight misnomer) allows a process to add licenses, to change its label downward, or to change its ceiling upward. Only `session` and `priv` have setlic capability.

`p` Setpriv privilege allows a process to change the privileges of files, usually programs. Only two programs have it.

Self-licensing programs

We have seen that a program gets a privilege if its process is licensed for that privilege and the executable file has the capability for that privilege. Executable files may also be ‘‘self-licensing.’’ The `su` command is one such file.

```
$ getlab /etc/su
/etc/su                 -u-n-- -u-n--    0000 0000 ...
```

The first set of privileges in the printed label are capabilities, the second set are licenses. As one might expect, `su` is self-licensed to write in the uarea (`u`) so it can execute the `setuid` system call. But why does it have nocheck privilege (`n`)? The reason is administrative. Customarily `su` keeps a console log. The console, like all ports, is labeled NO; `su` has nocheck privilege to bypass the label check.*

Another self-licensing program is the priv server, which needs setlic capability (`l`) to issue licenses. The priv server is a permanent program; `/bin/priv` simply passes it information.

```
$ getlab /etc/priv /bin/priv
/etc/priv               ----l- ----l-    0000 0000 ...
/bin/priv               ------ ------    0000 0000 ...
```

The capability field of a running program describes its actual privileges. The license field tells what licenses it holds to be passed on across `exec` system calls. Self-licensed privileges are not passed on. The session program, for example, has three capbilities and is self-licensed for two of them.

```
$ getlab /bin/session
/bin/session            --xnl- --xn--    0000 0000 ...
```

The `session` program checks authorization, using nocheck privilege (`n`) to access the secret `pwfile`. It then forks. The child process uses extern privilege (`x`) to set the terminal label. As long as the process label stays between floor and ceiling and the ceiling doesn’t go up, `session` can work for itself, without going through the priv server or the black hole. Otherwise it requires set license privilege (`l`), the license for which comes from the priv server.

What label would `ps` have to have in order to examine the core image of `session`? At least as high a label as that of the data `session` has read, namely `pwfile`. Nocheck privilege permits reading regardless of the label of the `ps` process. With no information as to the real label of the information the process contains, the system assumes the worst and assigns the process file a top label. Only a top-labeled process, or another nocheck process, can inspect the image of any process that ever had nocheck privilege. Similarly a core dump of a nocheck process gets a top label.

Trust

Any program with privileges is called ‘‘trusted.’’ The integrity of the system depends as critically on the honesty of trusted programs as it does on that of the kernel. Programs must be carefully written and checked before being granted trust. Moreover trusted programs must not change. The only changes that can be made to a trusted program are changes in its trustedness, and those changes themselves require privilege.

No matter how carefully it is written, a program can’t be trusted more than the data it receives as

---

* An astute reader will see that the console log can’t be public, for secrets can be sent to it this way:

```
$ /etc/su attack_at_noon
```

input. For that reason password checks are made over ‘‘private paths’’, so that unintended processes can neither eavesdrop on nor corrupt the exchanges. A private path connects a ‘‘trusted source’’, usually a secure terminal, to a trusted program. No untrusted program may intervene in a private path. On a `mux` terminal, the existence of a private path is marked by checkered border around the pertinent window. If you don’t see the border, you know that somebody is spoofing you.

Similarly, the priv server, which usually receives requests from an untrusted shell, uses a private path to confirm a request before acting on it. A program such as `/bin/setlab`, which can do magic only when it inherits a license from its invoker, will not usually check its arguments, however. Instead it understands that the license could only have come from a trusted program that has already guaranteed the integrity of the input.

Nosh

As we have already pointed out, standard shells are untrustworthy. Besides abounding with hidden and ill-described features, they are programmable, which means that no matter how perfectly a shell is implemented, the current meaning of any shell command is unknown.

For delicate situations IX provides `nosh`, the no-feature shell. This shell is used for the startup script, `/etc/rc.nosh`, which plays the same role in IX that `/etc/rc` does in ordinary UNIX. The `nosh` shell is also used for sessions below the floor, which can be obtained only with privilege.

```
$ /etc/priv session -l 0
Password(you:57146):
priv(session -l 0)? y
$$
```

The prompt changes to `$$`, the signature of `nosh`.

To avoid surprises, `nosh` has no search path.

```
$$ echo hello world
first letter not / or .
$$ /bin/echo hello world
hello world
```

It does, however, let you change the working directory.

```
$$ cd /bin
$$ ./echo hello world
hello world
```

If `nosh` is invoked with privilege, the prompt reminds you which ones are available. Authorized users can get a privileged shell from the priv server. (In practice they don’t; the only time a privileged shell is routinely invoked is in single-user mode at boot time.)

```
$$ /etc/priv nosh xn    # ask for xn licenses
Password(you:52892):
priv(nosh xn)? y
xn$$ /bin/getlab
proc lab                ------ ------    0000 0000 ...
proc ceil               ------ ------    ffff 0000 0000 ...
```

The `getlab` report reveals no privileges even though the prompt did. This is more paranoia; to avoid licensing a program inadvertently, `nosh` will not pass a license to a command unless you ask it to by supplying a ‘‘license mask.’’

```
xn$$ lmask n /bin/getlab
proc lab                ---n-- ---n--    ffff 0000 0000 ...
proc ceil               ------ ------    ffff 0000 0000 ...
```

The second `n` reports the license; the first `n` reports that `getlab` has nocheck privilege. The first `n` results

from and-ing the licenses with the capabilities of the executable file. It happens that `/bin/getlab` has nocheck capability, so that authorized users can use it to see forbidden device labels.

```
xn$$ lmask n /bin/getlab /dev/dk/dk12
/dev/dk/dk12  [name]    ------ ------ RN 0000 0000 ...
```

The device is rigidly (`R`) labeled NO (`N`), as it ought to be.

Control-D returns from the privileged `nosh` to the low session—still using `nosh`. The device label is no longer visible because the shell has no license to give to `getlab`.

```
xn$$ <control-D>
$$ lmask n /bin/getlab /dev/dk/dk12
/dev/dk/dk12: Security label violation
  e=001
```

The first error comment comes from `getlab`. The second, from `nosh`, reports the exit code returned by `getlab`.

In the low session, higher-labeled data are visible. Label controls, however, prevent their being written to the low-labeled terminal.

```
$$ /bin/ls
  t=015
```

The command terminated abnormally with termination code octal 15, `SIGPIPE`. The reason is that the label of `ls` rose to the floor as it read the current directory. To prevent high-labeled data from reaching the bottom-labeled terminal, the system discarded the output of `ls` and killed the process just as if it had written on a broken pipe.

Multilevel Windows on a Single-level Terminal†

M. D. McIlroy

J. A. Reeds

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Outboard from the IX system described in a companion paper, ‘‘Multilevel security with fewer fetters,’’ are ‘‘intelligent’’ terminals that contain a local operating system to support multiple windows and downloaded programs, all without benefit of memory management hardware. A program in the host mediates between (multiple) shell sessions and the terminal. To run multilevel windows, the host program needs to run as a privileged program, keep track of labels, and monitor the trustedness of the terminal. Very small changes in the terminal program enforce mandatory security policy.

Mux, the manager of *layers*, or windows, for Teletype 5620 and related terminals,[1, 2] poses difficult security problems. In principle, each layer on a terminal behaves as a separate virtual terminal serving its own shell and associated process group. To get the most out of the terminal, we wish to run each layer with a separate label. Hence data transfers among layers must obey the formal security policy. This is difficult because layers are mutually accessible. It is possible to copy data between layers. Worse, it is possible to download arbitrary programs into layers, and layers enjoy no hardware protection.

Mux is implemented by a pair of programs, a host part that multiplexes data transfers to the terminal and a downloaded terminal part—a multiprocess operating system in its own right. The host part multiplexes bidirectional traffic to all layers. Since it must deal with layers that have different labels, the host part must be trusted, with capability *Tnochk*. The host part deals with the process group of a layer through a pipe, which the process group sees as a terminal. The pipe obeys the same labeling discipline as would a terminal; its label is marked rigid and can be changed only by trusted processes with T_{mount} capability. To detect label changes the host process enables signal `SIGLAB`. It accepts all changes, secure in the knowledge they they must have been made by trusted processes. Each change is relayed to the appropriate layer. Upon a downward label change, the layer is reset to expunge all extant data; in effect the process group gets a new layer.

The terminal part knows only enough about labels to prevent leaks. It does not implement the full dynamic label scheme. Labels are checked on every attempt to cut and paste data between layers; attempts to copy downward are ignored. As an extra precaution in the face of a shared address space, the terminal part erases all storage as it becomes free, including screen bitmaps, downloaded programs, and displayed text. No logging of actions at the terminal is provided, however.

Programs to be downloaded into layers run in the native hardware of the 5620 and have access to the entire address space of the terminal. Hence code run in the presence of multiple labels must be trusted. An untrusted program may be countenanced only if its label is identical to the label of all data in the terminal; otherwise it could write down to or read down from other processes. Moreover an untrusted program cannot be loaded into the terminal while any private path reaches the terminal. Since nothing can prevent untrusted code from modifying the terminal part of *mux* itself, the terminal, which becomes untrusted as

† An earlier version of this paper appeared in *Proceedings UNIX Security Workshop*, Usenix Association, Portland, August 1988, 24-31.

soon as untrusted code is downloaded into it, must remain untrusted forever – or until rebooted. The labels stick at the untrusted value. In practice we are more stringent and require all labels in an untrusted terminal to have the same value as the the initial label of the terminal.

To separate concerns, *mux* was designed so that downloading would be done through it, not by it. Yet, to trust the terminal, *mux* must assess the trustability of downloaded code. Downloads into layers are handled by a trusted specialist program, *32ld*, that passes data through *mux*. *Mux* ascertains the trustedness of the downloader program and its connection to *mux*. The downloader is expected in turn to determine the trustability of downloaded code. Since the main pipe between *32ld* and *mux* is shared by a shell and perhaps by other processes, we protect communications over that pipe by using pex to prevent other processes from using the pipe until its status reverts. The downloader is deemed trustworthy if it has capability T_{mount}, which it relinquishes when downloading an untrusted file. *Mux* observes the trustedness by examining indicia of privilege that come with pex. Unless the downloader is trusted, *mux* marks the terminal untrusted.

A legitimate trusted download ends with a special coda from *32ld* that must be received while the pipe is marked for exclusive use. If an imposter kills *32ld* in mid-download the download pipe becomes unusable: the *mux* end is marked for exclusive use, while the other end reverts to permissive use. This state prevents all IO activity, in particular, attempts by the imposter to forge a download or to reassert process-exclusive access. *Mux* detects the change in state with a failed *read* system call and deletes the layer. The terminal remains trusted.

The standard version of *mux* depends on *32ld* to download the terminal part before the host part begins. We have abandoned this arrangement, which made it difficult to assure the host part that it is indeed talking to the correct terminal part. Instead, we let *mux* do that download itself, again protecting its access to the terminal with pex. In fact, process-exclusive access is retained through the whole *mux* session. This curious division of labor, wherein downloading into the raw terminal and into layers is done by different agents, is marginally justifiable: existing programs that need to load into layers already know about *32ld*, and the protocols differ, one being in hardware and one in software.

Mux also uses pex to provide a private path between user terminal and application program. While the terminal is trusted, a trusted user process running in a layer may issue a pex call on its pipe to *mux*. As in the special case of *32ld* sketched above, *mux* detects the new process-exclusive status of the pipe together with indicia of the privilege of the process that issued the call. It notifies the terminal part, which in turn gives the layer a distinctive visual mark. The user then knows that new data typed in that layer are not accessible by eavesdropping programs. This private path mechanism can be used for confidential negotiations, such as password collection, that should not pass through an untrusted terminal.

Various flaws remain.

First, the 5620 itself might be subverted by downloading a terminal simulator that could receive and corrupt a later download of *mux*. This is impossible to do without giving a visual indication on the screen, but it could happen while nobody was looking. One way to detect such a gambit would be to download a program that fills memory with hard-to-compute numbers, then answers inquiries about the contents of randomly chosen locations. If it doesn't answer fast enough, the memory may be suspected of harboring other, unwanted, code. Another countermeasure would be to modify the hardware to provide an out-of-band channel from the host to the native boot program. We have taken neither precaution, counting instead on users avoiding the trap by booting the terminal afresh just before starting *mux*. Aside from its dependence on human behavior, this procedure is sound: if a phony *mux* were downloaded instead, it would not be trusted, so multilevel data could not be accessed.

Second, under *mux*, the terminal practically becomes an extension of the operating system. For genuine safety, the physical security arrangements for the host computer should be extended to every *mux* terminal, and especially to the terminal's ROM.

Third, the private path for session-authorizing negotiations outlined above, will not work with an untrusted terminal. This is only a special case of a more general question: how to perform an authorizing negotiation over any external medium, the trustedness of which has not been established, and may be unnecessary for the session that follows. Challenge boxes, which accomplish confidential negotiations in the presence of eavesdroppers, are the preferred solution.

REFERENCES

[1] Pike, R., ‘‘The Blit: a multiplexed graphics terminal,’’ *Bell Laboratories Tech. J.* **63**, pp. 1607-1631 (1984).

[2] AT&T Bell Laboratories Computing Science Research Center, *UNIX Research System Programmer’s Manual*, Vol. 1, Saunders, Philadelphia (1990).

Secure IX Network†

Jim Reeds

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper sketches a design for a network of computers running the McIlroy-Reeds IX system. The emphasis is on modularity and decentralization; security does not rely much on central key distribution. It assumes that there are multiple overlapping domains of authority, and relies only loosely on an ultimate common organizational loyalty. This work is speculative. It is heavily influenced by the networking arrangements in the Research 10th Edition UNIX® system.

1. Single Machine IX*

When we began working on IX M. D. McIlroy and I simply wanted to add to the usual UNIX system model a stricter file access policy, a variant of the military security classification system. We believed that a no-frills implementation of this part of the Orange Book[DOD] requirements for a secure computer would satisfy most real security needs of most users, even if it might NOT satisfy the Orange Book people themselves. We also believed that our new access restrictions could be kept largely orthogonal to the existing scheme of `rwx` bits, which we left essentially intact. The new access policy could be airtight and draconian in its preservation of security labels (and the intellectual property rights they represent), even if on the same machine, at the same time, the usual UNIX system concepts of userid, setuid, root accounts, and `rwx` file permission bits were subverted.

After three years of part-time but intense work we have not changed those beliefs, but we have expanded our notion of what needs to go into a ''no-frills implementation.'' Our resulting system, like all other attempts at secure operating systems, ends up with a two-tier structure. There is the generality of user programs, including all programs written by ordinary users and most of the programs in the public program directories. And then there are the privileged programs used to administer the security system, covering functions like logging users in, changing user's security clearances, and so on.

The distinguishing property is that the privileged programs must break the usual security rules, or viewed another way, that the kernel must be assured by a privileged program that some particular violation of the usual rules is not in fact a breach of security. Files, for instance, are not usually allowed to drop in security classification level. Terminal ports are files in the UNIX system, so when a terminal port used in a classified login session becomes free at the end of the session, before it may be used for a new, unclassified, login session it must be declassified by a privileged program. Ordinary disk files might be declassified from time to time for the usual reasons; this again needs a privileged program. The kernel cannot itself know when such actions are acceptable: it relies on the privileged programs to tell when to deviate from its worst-case application of the usual rules.

---

† Reprinted from DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Volume 2, *Distributed Computing and Cryptography*, J. Feigenbaum and M. Merritt, editors, pp. 235-244, by permission of the American Mathematical Society.

* Some details of single-machine IX given below are, for exposition's sake, simplified. The true description is in the accompanying papers. IX is *not* pronounced like the German word *ich.*

Trusted Computing Base

The presence of these privileged programs complicates things enormously. Obviously questions like ‘‘how do I know that a baddie hasn’t tampered with a privileged program,’’ or ‘‘how do programs become privileged,’’ or ‘‘how do new releases of privileged programs get installed on the system’’ are relevant. The easy answers (essentially the same in all secure operating systems) are: Giving or removing privileged status to files can only be done by a privileged program. A privileged program can not be removed or changed, other than to have its privileged status removed. And the program that first runs at boot time, *init*, is privileged. The privileged programs thus form a self-nominating closed ‘‘committee’’ which, together with the kernel as an *ex-officio* member, runs the computer. The jargon for this committee is TCB: trusted computing base.

This in turn leads to the hard questions. How does the TCB know which information from the outside world it should act on, and which is spurious? How does the computer user know he is talking to the TCB and not an imposter? How does one privileged program communicate with another, in such a way that each recognizes the other’s privilege? IX’s answers to these questions are based on two mechanisms, one in the kernel, the other coded into the privileged programs, both of which seem easily extendible over a network. Hence the TCB’s iron reign can be extended over a network, and hence so can IX security services.

PEX

The first of these wonder mechanisms is a form of mandatory exclusive file locking, which we call PEX for ‘‘process exclusive.’’ Any file may be marked for exclusive use by a process in which it is open, so long as no other process has done so first. The exclusivity lasts until the process explicitly drops it or until the file is closed or the process exits. A special twist occurs in pipes, each end of which may be independently PEXed. A pipe will not carry data unless both ends are PEXed or neither end is PEXed: an attempt to read or write on a half-PEXed pipe result in an error. When a pipe end is PEXed the other end is given indicia of privilege of the PEXing process. All this is done synchronously and independently of the usual stream[R] discipline. Thus a privileged process communicating on a pipe can tell if it has a unique partner process at the other end, and whether the partner process is privileged. A privileged program, for example, can refuse to collect a password unless it can PEX the standard input. If talking to an IO device unPEXed to any other process, the PEX call succeeds and no interloping process that also has the device open can steal the password. If talking to a pipe (say, to a window on a windowing terminal) the PEX bid will be accepted or rejected by the process at the other end of the pipe. If accepted, the privileged program can now tell if the other end is also in the TCB (as a trusted terminal multiplexor running secure windows would have to be), in which case the password dialogue can go forward, and so on. Using these means parts of the TCB can recognize each other, can know when they are talking to users, and —although the details are clumsy— can prove to users that they are indeed talking to the TCB.

This can be thought of as a generalization of the Orange Book’s ‘‘trusted path’’ mechanism, and as an alternative to Biba-style ‘‘inverse labels’’. (Biba inverse labels are described in pages 69-71 of Gasser[G]; the labels track trustability or provanance of data rather than secrecy.)

The notion of a PEXed pipe is nothing other than that of a secure communications channel between identified parties, an end goal of both classical and modern cryptology and of secure network design. The novelty is in making this service available to user processes, protecting themselves from eavesdropping, forging or spoofing by other processes. In its current single-machine form, the writ of PEX ends at the machine boundary, and cryptography is not needed to implement this new operating system feature. (New to the UNIX operating system, at least.) For PEX to go over a network to another machine would require at least a special protocol for the kernels to use in discussing the PEXity of their ends of the cross-machine pipe, and also possibly cryptography to identify and authenticate the end machines and to protect the data in transit.

Privilege Server

The second special mechanism for privileged process control in IX is a *privilege server* embodying a particular design for denoting, recording, and exercising users' rights. The main idea is that although the TCB is all-powerful, it doesn't have any security interests of its own, save self-protection and economic control of the local hardware resources. The TCB is a custodian of users' rights, which might originate off-machine, possibly created under an authority separate from the management or ownership of the local host computer. In IX these fiduciary responsibilities of the TCB are managed by the privilege server.

The privilege server centralizes *authorization* calculations, much the way an authentication server centralizes authentication services. Authorization calculations often take a stylized form: a resource's owner passes a credential to a user, who later presents the credential together with a particular resource usage request to the custodian of the resource. In IX the privilege server is the custodian of most users' rights (other than routine file access rights implied by user id and process label). For each request, the privilege server must check that the credential indeed authorizes the request, that the privilege server itself is authorized to act on the request (which includes checking whether the resource in question has been entrusted to the privilege server), as well as authentication information, such as whether the issuer of the certificate is the owner of the resource, and so on.

Requests for privileged services are made to the privilege server as text strings, which are also statements in a trivial computer command language, *nosh*. (This is a restricted, feature-starved ''secure'' shell with fewer possibilities for latent subversive side effects than found in the usual interactive shells *sh*, *csh*, or *ksh*.) If the requester has a right warranting the command string in question then the privilege server executes the command. Corresponding to each right, then, is the set of *nosh* command strings it warrants.

More precisely, a ''right'' is a regular language. Rights are recorded in a privileges file as pairs (R, Q), where R is a regular expression* (specifying the set of *nosh* commands warranted by the right) and where Q is a predicate applied to: the user, the execution environment of the current invocation of the privilege server, the status of the user's connection to the server, and (optionally) a protocol execution status. When a request s is presented to the server, it is executed only if a pair (R, Q) can be found such that s is an element of R and such that the predicate Q evaluates true. Thus ''anyone'' who satisfies Q may exercise R.

For ease of subdivision and delegation, we organize our privilege file as a rooted tree. Node (R_1, Q_1) may be above (closer to the root than) node (R_2, Q_2) only if language R_2 is contained in R_1. Thus sub-nodes correspond to sub-rights, and if you can exercise a right at a given node you automatically may do anything you could do by exercising rights at sub-nodes.

Our command language has statements to edit the privileges file, so it is possible to formulate rights to change rights. In particular, it is easy to formulate a right to edit only a sub-tree of the tree of rights. Allowed edit commands are: diminish the set of rights at a node, delete a node or sub-tree, create a sub-node with rights equal to the parent node, and change the Q part of a node. Thus editing can only reparcel or dilute existing rights, but not create new ones. (Since privilege file edits are relatively infrequent there is no practical performance loss in forbidding concurrent edits, or forbidding privilege calculations during edits.)

To delegate a right means simply to create a sub-node in the tree of rights.

The Q predicate might require given login ids, ask for a password, require successful completion of a challenge-response protocol, or insist that the privilege server is being invoked by a particular named program. Q specifies the *authentication* needed to enjoy the rights in question.

A weak point in this scheme lies in the formulation of rights as regular expressions: extreme care must be taken to ensure that the *nosh* command language statements accurately catch the real meaning of the rights in question. In particular, the manual describing the TCB and the *nosh* language must be accurate, and worse, the meaning of command-line arguments and options to privileged programs may never be lightly changed, lest established rights be overthrown or inadvertently widened.

---

*Abuse of notation to follow: I make no notational distinction between a regular expression R and its corresponding language, so sometimes R really means $L(R)$.

An Example

Here is how the privilege server helps administer a security classification compartment. Suppose some users of an IX machine start a new project and want to create a security compartment to guard their work on the computer. This new security compartment shows up outside the computer as a new classification keyword stamped on documents and inside the computer as a new bit field in the security labels carried by files and processes. Say project leader Alpha wants to create a compartment `BETA` for his project; what steps does it take to teach the TCB about this new compartment?

To begin with the TCB's only interest in `BETA` is in issues of operating system resource exhaustion: is the keyword `BETA` already in use for some other security compartment and is there an unused label bit field on the local machine to represent `BETA`? The usual solutions to these problems apply: categories might have hierarchical names (keywords like `ADONIS.PICCOLO.BETA` are less likely to be preempted), and only certain users may consume label bit fields by creating new security classifications. So among the rights already stored in the privileges file is an entry $(R_{\text{mkcat}}, Q_{\text{mkcat}})$ giving project leaders the right to run the privileged *mkcategory* program:

$$R_{\text{mkcat}} = \texttt{mkcategory .*}$$

$$Q_{\text{mkcat}} = \textit{login\_id} \in \{\textit{Alpha}, \textit{Rocky}, \textit{Boris}, \textit{Natasha} \cdots\}.$$

So Alpha presents the request

```
mkcategory BETA
```

to the privilege server, which runs the *mkcategory* program in such a way that *mkcategory* knows this invocation is an authorized invocation. Note that so far the privilege server has only been guarding the local machine's operators' interest in preventing resource exhaustion, and not any security interest *per se*.

Mkcategory has two functions. One is to allot a bit field in the machine's internal representation of security labels and bind the name `BETA` to it, registering this in an official list. The other function is to register Alpha as the ''owner'' of `BETA` by placing a new entry $\beta = (R_\beta, Q_\beta)$ in the privileges file. (The *mkcategory* program itself has the right to edit part of the tree of rights, according to a $Q_{\text{mkcat2}} = \textit{invokerprog} \equiv \texttt{mkcategory}$.) To begin with *mkcategory* finds out from Alpha what formula to use for Q_β, that is, how future commands from Alpha about `BETA` will be recognized. Alpha may provide any formula for Q_β he wishes: he may simply specify a password, or may specify that possession of Alpha's *login_id* suffices, may present the public key by which future commands may be verified, *a la* RSA, and so on. The *R* part of the new right is always the same for a newly created category, and expresses the union of these rights:

> Exerciser may access data marked `BETA` by logging on with the appropriate bit set in his process label.
>
> Exerciser may declassify category `BETA` from files, by executing a privileged command of form `downgrade BETA .*` which clears the `BETA` bit from the named file.
>
> Exerciser may specify network addresses that may receive `BETA` data.
>
> Exerciser may edit this node β in the tree of rights.

Now project leader Alpha has a private security classification label all of his own. He can make classified login sessions, he can create secret files readable by no one else, he can declassify them.

Soon however Alpha wishes his assistants Mr. Gamma and Ms. Delta could access his secret files, too. He invokes his right to edit node β (by presenting whatever credentials he earlier spelled out in Q_β) and creates a sub-node, call it β*/access*, whose *R* part consists solely of the exerciser's right to read and write data marked `BETA`. The *Q* part is again at Alpha's discretion, who (say) chooses the formula

$$Q_{\beta/\text{access}} = \textit{login\_id} \in \{\textit{Gamma}, \textit{Delta}\} \;\&\; \textit{shows\_password}(\texttt{cyto97plasm}).$$

(Alpha need not add himself to the list in $Q_{\beta/\text{access}}$ because he already may exercise the superior right β, but might want to do so because for simple file access $Q_{\beta/\text{access}}$ might be easier to use than Q_β.) Note that Alpha has extended only his access rights to Gamma and Delta: as desired, Gamma and Delta can read and write `BETA` secrets, but they cannot change the list of people so cleared.

The project prospers, and when new assistants Eps through Lamb show up boss-man Alpha tires of editing β*/access*. So he appoints Ms. Delta as his BETA-clearance officer, by creating a new node β*/clearance*. $Q_{\beta/\text{clearance}}$ only lets Delta in. $R_{\beta/\text{clearance}}$ only allows the exerciser to edit the predicate $Q_{\beta/\text{access}}$.

At appropriation time some good press coverage is needed, so Alpha appoints Gamma his publicity manager, whose job of course includes shrewdly sequenced leaks of BETA data, which means a new node β*/declass* is formed, with a $Q_{\beta/\text{declass}}$ for Mr. Gamma alone, and with $R_{\beta/\text{declass}}$ allowing BETA-downgrade rights to the exerciser.

What would it take to extend these activities to another computer? Obviously at set-up time the remote file servers need to reconcile their differing internal representations of file security labels: it is almost guaranteed that the bit slot allotted to represent category BETA will differ on all machines. But before BETA files may be traded the two machines should really check to make sure that what each knows as category BETA is really Alpha's BETA, by trading and verifying credentials signed by Alpha, according to the recipe spelled out in Q_{β}.

2. Networked IX

The general shape of a network of IX machines is clear. If a pair of machines trust their network connection, and trust each other's TCBs to enforce the same basic security policies, then they may extend PEX service and remote file system service as sketched above. A minimal version might consist of the following ingredients:

- A LAN offers secure networking within the confines of a single department.
- The individual IX machines are run by the same systems staff, who run identical software on the machines, so any of the machines can trust any other's TCB as much as it trusts its own
- Terminals are at known network locations at known physical locations

Within such a network different machines can carry varying mixes of secret data: some label categories may exist on all machines, other categories may exist on single machines only, other categories yet may exist on other subsets of machines. The user communities may be heterogeneous: not all users may have accounts on all machines, nor need all users of a given machine have access to the same security categories. Such a setup might be appropriate in a small department, or in a rigidly run branch of the government.

A more ambitious network might include subnets of the above description but would also have insecure network links, a variety of terminals in unknown or uncontrolled locations, a variety of computers with differing software, some of which are in isolated secure physical locations. Such a network would need a measure of cryptographic boost to give untappable connections, or connections to known endpoints. While it is unreasonable to hope that the computers all run exactly the same system software there is still a chance that their security software is the same. If two machines each believe the other's TCB there is a possibility that they can trade security services. To prove that partner machines have correct TCBs some appeal to authority is needed. Depending on details, machine A might trust B if B is at a known network address, or if B has a document attesting B's honesty, signed by an authority A trusts, or (what is really a variation of this) if B can communicate at all with A when A uses cryptographic keys it was provided with by a trustworthy authority.

In an ambitious network user terminals and user authentication pose a real problem: there is no general way for a network or a computer to tell the difference between a terminal and a work station or heftier computer. Any command purporting to come from a user at a terminal might really come from a hostile computer, which can play the IX protocols for dishonest purposes. Terminals on the network are thus treated as computers: unless they are at certain special secure network addresses on a trusted network, they must prove their *bona fides* before they are given PEX service, and hence before they may be used to get favors from the privilege server.

Little special software is needed for networked IX given single-machine IX and given regular UNIX system networking facilities. As hinted earlier, a protocol for the TCBs to manage cross-machine PEX would be needed, possibly as a stream [R] module. And to handle remote file systems, for example, a form of ''power of attorney'' protocol is needed, by which one machine can prove to another that it has authority

from a third party known to both. To be usable there must be user software to make it convenient to create (say) RSA keys on demand, and so on.

3. Special Terminals and User Authentication Hardware

In a network of IX machines, terminals at unknown or uncontrolled network locations must be treated as potentially hostile computers, which of course makes logging in, authenticating users, and reading secret data over a terminal difficult. Here is a possible solution to these problems, relying on special hardware, prompted by our experience with windowing terminals in single-machine IX

The main idea is to use special purpose secure computers as terminals, together with smart-card-like user identification tokens which are also special purpose secure computers.

The terminal and screen software is part of the TCB. The TCB is unprogrammable (in ROM, say) and contains secret cryptographic keys in an uninspectable store; the whole packaged in a tamper-proof container. The TCB can enforce a primitive form of IX file access policy (each multiplexed terminal process and associated windows, say, has a label, with label inequalities obeyed on mouse-initiated cross-window copies or ‘‘snarfs’’). Also, a form of PEX is available: if a screen layer is accessible by exactly one terminal process, and if the keyboard is accessible only by that terminal process, then a distinctive unforgeable visual mark may be placed on the screen layer (a flashing border, say).

The ‘‘user authentication tokens’’ are small special purpose computers with their own TCBs and tamper-proof memory. A token can be plugged into a special terminal, and has a light visible to the user when the token is plugged into the terminal.

This hardware is used to help several authentications. There are as many as four parties with differing security interests: user, user token, terminal, and host computer. The user token must be convinced that its legitimate user is present. The terminal and host must like each other’s TCBs enough to set up cross-machine PEX. The host must communicate through the terminal to the user token, but the terminal will not play PEX unless it trusts the token, and so on.

Most of these security interests are satisfied by multiple applications of, say, the Fiat-Shamir[FFS] protocol, where the terminal, host, and token successively play differing roles. The FS protocol has two roles: the ‘‘prover’’, who has a certificate issued by an authority, and a ‘‘verifier’’ who checks the certificate without—and this is the big trick—actually seeing it. The authority knows the factorization of a modulus N. The modulus N, but not its factorization, is known ahead of time by the verifier. What the verifier knows at the end of the protocol is that whoever prepared the certificate knew the factorization of N.

The terminals and tokens carry certificates from their manufacturers, signed according to a modulus N_{term}, characteristic of the brand name of the terminal. This modulus N_{term} is known by all the host computers. This prior distribution of public authenticating key is not a burden because the number of distinct brands of secure terminals will remain small.

User identification tokens also carry certificates attesting to the identity of their owners. These certificates are signed with a modulus N_{dom} characteristic of the security domain in which the user lives. There is nothing wrong with a user identification token carrying several such certificates valid in different domains. Computers also have one or more certificates signed by the same moduli N_{dom}. The assumption is that the same authorities that certify users are able to certify computers. Roughly, if you have an account on a machine, there is at least one N_{dom} you and the machine have in common.

Here is how to set up a login session with such equipment. First, the token and terminal authenticate each other with respect to modulus N_{term}. They tell the user that they like each other by visual signals: the light on the token is lit, a special icon or message is displayed on the terminal. Then the token has a chance to demand a password from the user, typed via the keyboard of the now-trusted terminal. Then the token tells its N_{dom} values to the terminal. Terminal and host computer negotiate to find a common N_{dom}, then the computer proves its *bona fides* to the terminal using modulus N_{dom} and the terminal proves to the computer that it is a special terminal with modulus N_{term}. The computer and terminal now set up the cross-machine PEX protocol on their link, and the user proves his identity to the computer either by typing a classical password or by letting his token prove the user’s credentials using N_{dom}.

A simplification is possible in the case where there is only one security domain which knows about

all trustable computers. Then the user token is not needed: the single modulus N_{term} can suffice for all applications of Fiat Shamir.

Conclusion

The forgoing describes a general architecture for a network of secure computers, offering far more security than is commonly found in the UNIX system, offering users a measure of distributed security services with a minimal overhead of centralized bureaucracy.

How large an organization could such a net serve, before the methods of user authentication, delegation of rights and authorities, and system administration sketched above becomes too cumbersome? How much work would it take to build such a net, given the existing pieces?

I have had chats with Baldwin, Coutinho, Feigenbaum, Fernandez, Fraser, Grampp, Kurshan, McIlroy, Merritt, Pike, Presotto, Ritchie, Thompson, Wilson, Zempol, among others, to whom I am grateful for ideas and helpful criticism.

References

[DOD] Department of Defense Computer Security Center. *Department of Defense Trusted Computer System Evaluation Criteria.* US Department of Defense, Fort Meade, MD, 15 August 1983.

[FFS] U. Feige, A. Fiat, and A. Shamir. ‘‘Zero-Knowledge Proofs of Identity’’, *Journal of Cryptology*, **1**:77-94, 1988.

[G] M. Gasser. *Building a secure computer system.* New York, Van Nostrand Reinhold, 1988.

[MR2] M. D. McIlroy and J. A. Reeds. ‘‘Multilevel Security with Fewer Fetters’’, in *UNIX Around the World: Proceedings of the Spring 1988 EUUG Conference.* European UNIX Users’ Group, London, 1988.

[MR3] M. D. McIlroy and J. A. Reeds. ‘‘Multilevel Windows on a Single-level Terminal’’ in *Proceedings, UNIX Security Workshop, August 29-30, 1988. Also in the present collection.* USENIX, Portland, OR.

[R] D. M. Ritchie. ‘‘A Stream Input-Output System’’, *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, October 1984.

GLOSSARY

This glossary defines terms peculiar to IX. The glossary for the Unix Research System, 10th Edition, which is incorporated by reference, defines certain terms used here: *argument, executable file, file, groupid, inode, kernel, permission, process, stream, superuser, system call, terminal, u-area, umask, userid, utility.*

accept pex indicator a control, set with *privilege* [1], on a stream to permit or deny *pexing* according as the stream is or is not *trusted* [3].

assured path a channel comprising *trusted* streams and processes that is understood to pass information faithfully without tampering or eavesdropping.

audit to record security-related events, such as file accesses, process creation, and exercise of *privilege* [1].

audit mask a bit vector associate with each process to specify the intensity of *auditing.*

bottom see *lattice label.*

capability 1. actual right of a process to exercise a *privilege* [2]; cf. *license.* Process capabilities, which can be relinquished at any time, are determined at *exec*(2), either by intersecting its licenses and the *capabilities* [2] of the file it is executing or by *self-licensing.* 2. potential right of an executable file to exercise privilege.

ceiling a *label* [1], which must dominate the label of any file involved in a system call. Every process and every file system has a ceiling.

constant see *fixity.*

covert channel an information path between untrusted processes that does not obey the *mandatory security policy.* Always of low bandwidth, covert channels usually involve inferences from error returns rather than *data flows.*

data flow explicit transfer of bits from place to place by system calls. Pertinent places are processes, files, directories, inodes, seek pointers, and u-area data, such as process *ceiling,* exit status, umask, userid, and groupid; cf. *covert channel.*

domination a relationship among *labels* [1]. A *lattice label* is said to **dominate** another if and only if the former has one bits in all positions that the latter does. A label with label flag value *yes* dominates and is dominated by any label. A label with *label flag* value *no* does not dominate and is not dominated by **no** or by any lattice label.

downgrade to change, by use of *privilege,* the lattice label of a file to a lattice label that does not *dominate* the previous value.

drop 1. to change the value of a process *label* so that the new value does not *dominate* the old value. A process label can drop only at *exec*(2) with no arguments. 2. to decrease the *ceiling* of a process, as by *drop*(1).

extern a *privilege* [2] that allows the *label* [1] of an open *external medium* to be set away from its quiescent value of **no**.

external medium a file, such as a terminal or magnetic tape, that communicates with the outside world. Because the *mandatory security policy* cannnot automatically be assured on external media, *privilege* [2] is required to initiate input/output thereon.

fixity the degree to which a *label* [1] on a file or process may be changed. The values of fixity are: **loose,** freely changeable to a dominating value; **frozen,** changeable only explicitly by the owner; **rigid,** changeable only with privilege; and **constant,** not changeable.

floor a conventional *lattice label* [1] assigned to a user's shell process at login. The floor is the label of the file `/etc/floor`.

frozen see *fixity.*

label 1. a designation of the *mandatory security* status of a file or process. 2. the representation of a label [1], comprising: *label flag, fixity, lattice label, capabilities* [2], and *licenses* [2].

label flag part of a *label* [2] that tells whether the label's value is a *lattice label,* or one of two special values, *yes* for generally readable and writable data, such as `/dev/null`, or *no* for generally unreadable and unwritable data, such as *external media.*

lattice label a designation of security level, the lattice label comprises 480 bits. Data flow is permitted only if the lattice label of the destination *dominates* the lattice label of the source. Lattice labels of all zeros and all ones are called **bottom** and **top** respectively.

license 1. potential right of a process to exercise a *privilege* [2]. A license can be relinquished at any time and is inherited across *exec*(2). 2. an indicator of *self-licensing* of a file.

log a *privilege* [2] that allows querying and changing the intensity of *auditing.*

log file a special file for *audit* information. A log file can be written regardless of labels and can be read by no process. Audit files are associated with ordinary files by *setlog*(2).

loose see *fixity.*

mandatory security policy rules to govern *data flow* regardless of 'discretionary' user decisions about file

permissions. Except on certain actions of *trusted* processes, a security *label* of the destination of any data flow must *dominate* the label of the source. Labels are calculated at every system call and are adjusted as necessary to preserve dominance. cf. *covert channel* and *TCB.*

no a non-*lattice label* that neither dominates nor is dominated by any *label* [1] other than *yes* . Because a file labeled *no* cannot be read or written by any un*trusted* [2] process, it is safe to set a file label to **no**; cf. *extern.*

nochk a *privilege* [2] that allows a process to access a file regardless of *domination.*

pex to assert process-exclusive access to a file. A pipe pexed at one end can be used only if it is also pexed at the other; see *pex*(4).

poison class a file attribute, visible and settable only with *privilege* [1], that forces auditing to at least a specified *poison mask* level when a process mentions the file.

poison mask one of several auxiliary bit vectors that can augemnt the *audit mask.*

privilege 1. mechanism of *capabilities* and *licenses* for controlling deviation from the basic *mandatory security policy* and for administering privilege. 2. one of six distinct classes of privilege: *extern, log, nochk, setlic, setpriv,* and *uarea;* cf. *trusted.*

privilege server the utility *priv*(1), which, following rules in the file *privs*(5), grants *licenses* [1] needed to exercise *privilege.*

rigid see *fixity.*

self-license possession by a file of a *capability* [2] and a corresponding *license* [2]. Self-licensing gives the corresponding *capability* [1] to a process at *exec*(2).

session an interval of running with special rights, usually evidenced by a distinct terminal *label* [1], *ceiling,* or *stream identifier;* see *session*(1).

setlic a *privilege* [2] that allows the *licenses* [1] or *ceiling* of a process to be set arbitrarily.

setpriv a *privilege* [2] that allows changing the *capabilities* [2] and *licenses* [2] of files.

stream identifier a string that is by exercise of *privilege* [1] attached to a stream to describe properties of the stream and its destination; see **FIOGSRC** and **FIOSSRC** in *stream*(4).

TCB, trusted computing base the kernel, *trusted* [1] utilities, critical data for these utilities, and utilities that may be used to process files in the TCB. Faithfulness to the *mandatory security policy* depends on the correctness of the TCB.

top see *lattice label.*

trusted 1. having some *capability* or *license;* said of a file, especially an executable file. The only way a trusted file can be modified is to change its privileges with capability *setpriv.* 2. having some capability; said of a process. Superuser processes are not necessarily trusted. 3. understood to be immune to tampering or eavesdropping, said of a stream associated with an *external medium;* cf. *assured path.*

trusted computing base Same as *TCB.*

uarea a *privilege* [2] that allows changing userid, groupid, and logname in the per-process u-area. The privilege is required lest these items, being both readable and writable by untrusted processes, provide a means to violate the *mandatory security policy.* The permission mask (umask), and the process *ceiling* are protected by other means; see *exec*(2) and *setplab*(2).

yes a non-*lattice label* that dominates and is dominated by any *label* [1]. A file labeled **yes** can be read or written by any un*trusted* [2] process.

NAME

getlab – print security labels of files and processes

SYNOPSIS

`getlab` [`-d`] [*file* ...]

DESCRIPTION

If there is a *file* argument, *getlab* prints, in the style of *labtoa*(3), the security labels of the named files. Otherwise, *getlab* prints the security label of the process and the process ceiling, The option is

`-d` Also print labels of all open file descriptors.

If a (character special) *file* can be opened, and the labels of the file and the file descriptor differ, both are printed.

EXAMPLES

`getlab /dev/stdin`

Print the labels (file system entry and file descriptor) of the standard input.

`drop getlab -d`

Print the process label, preventing the open-file check and the ceiling-label check (see *getplab*(2)) from raising the process label.

SEE ALSO

stat(1), *getflab*(2), *getplab*(2)

NAME

sign, enroll, verify, key, notaryd – sign and verify certificates

SYNOPSIS

`notary sign`

`notary enroll` [`-n`] *name*

`notary verify` *name xsum text*

`lmask xn /usr/notary/notaryd` [`-m` *mtpt*] [`-d` *dir*]

`notary key`

DESCRIPTION

Notary provides a document-authentication service. Any user may 'sign' a document by presenting it and a secret key to the notary. The notary returns a certificate (a cryptographic checksum made with the secret key). For the certificate to be useful, the key must be enrolled with the notary under some public name. Given the certificate and the public name, any user may ask the notary to authenticate the document by verifying that it is indeed as certified.

Sign writes on the standard output a certificate for its standard input. The secret key is demanded from the terminal.

Enroll prompts the terminal for a secret key to associate with the public *name*. Unless this is a new enrollment for *name*, indicated by option `-n`, the previous value of the key is demanded from the terminal. If a trivial new key is presented, the *name* is erased from the database.

Verify tells whether *xsum* is the checksum of *text*, figured with the enrolled key for the public *name*.

Notaryd is the notary daemon, which mounts itself on *mtpt* (default `/cs/notary`) and keeps its log files and database in directory *dir* (default `/usr/notary`). The database is encrypted, so that although *notaryd* is normally started by *rc*(8), it cannot serve other requests until it has been primed by a `notary key` request, which obtains the notary's master key from the terminal.

FILES

`/cs/notary`
`/usr/notary/*`

SEE ALSO

notary(3)

NAME

passwd, pwx – change login password

SYNOPSIS

`passwd` [`-an`] [*name*]

`priv pwx` [[`-qcd`] *name*]]

DESCRIPTION

Passwd changes a password associated with the user *name* (your own name by default).

The program prompts for the old password and then for the new one. The caller must supply both. The new password must be typed twice, to forestall mistakes.

New passwords must be at least four characters long if they use a sufficiently rich alphabet and at least six characters long if monocase. These rules are relaxed if you are insistent enough.

Only the owner of the name or the super-user may change a password; the owner must prove he knows the old password.

If the `-a` option is given, *passwd* prompts for new values of certain fields of the password file entry.

The super-user may use the `-n` option to install new users. The prompts are self-explanatory, and most of the defaults obvious. A null response to the `UID:` prompt assigns a numeric userid one greater than the largest one previously in `/etc/passwd`. A null response to `Directory:` assigns a home directory in `/usr`. If the first character of the response to this prompt is an asterisk, the remaining characters are taken as the name of the new user's home directory, and a symbolic link to this directory is placed in `/usr`.

A new user's home directory starts with a file named `.profile`, which is a copy of `/etc/stdprofile` with `\N` replaced by the user's name, and `\D` replaced by the name of the user's home directory.

Pwx modifies the password entry for the named user in the secret password file, *pwfile*(5). With no option *pwx* changes the classical password for the named user, or the invoker by default. The options are

`-c` Change other information. A special editing password for a fictitious user, 'pwedit', is demanded. Then *pwx* prompts for treatment of the user password, SNK key, maximum privilege, and clearance (maximum ceiling).

`-d` Delete an entry. The editing password is demanded.

`-q` Demand the user password. If a correct password is entered, return status 0; otherwise nonzero.

Options `-c` and `-d` require `T_SETPRIV` privilege.

FILES

`/etc/passwd`
`/etc/stdprofile`
`/etc/pwfile`

SEE ALSO

crypt(3), *passwd*(5), *pwfile*(5)
Robert Morris and Ken Thompson, 'UNIX password security,' *AT&T Bell Laboratories Technical Journal* 63 (1984) 1649-1672

BUGS

The password file information should be kept in a different data structure allowing indexed access.

NAME

pcopy – paranoid file copy

SYNOPSIS

[`priv`] `pcopy` [*input output*]

DESCRIPTION

Pcopy copies an input file to an output file preserving, if possible, file ownership, dates, and label. The copying is performed in such a way as to assure faithfulness even in the presence of interfering processes.

Privilege, obtained via *priv*(1), is required to reproduce privileged files. The user must be able to write the output file, and be able to read and write files with the label of the input file.

SEE ALSO

cp(1), *pex*(4)

NAME

priv, privedit – run a command with privileges

SYNOPSIS

`priv` [*option ...*] [*command arg ...*]

`priv privedit` *node changes*

DESCRIPTION

If a *command* is given, *priv* determines from the *privs*(5) file the most specifically matching `REQUEST` for which the process has all the `NEEDS` and to which it has `ACCESS` (terminology explained in *privs*(5)). If a unique most specific match is found, *priv* asks for confirmation. Then, if the confirmation is `y`, the request is executed. Privileges and process ceiling are set according to the pertinent entry in `/etc/privs` and the current directory is set to a place with security label `L_NO`; see *getflab*(2). Thus relative pathnames won't work in the *command* until it executes *chdir*(2).

If no command is given, the contents of the *privs* file are printed on the standard output.

The options are

`-n` Determine and report authorization and actions. Do not execute them except, if `PRIVEDIT` is requested, place the edited privilege file on the standard output.

`-f` *servfile*
Use *servfile* instead of `/cs/priv`, to use a non-standard privilege server.

One request is more specific than another if the regular language for each argument of the first request is contained in the corresponding language for the second request, and at least one containment is proper.

The standard error and standard input are used for confirmations. Both must come from the same trusted source, either a pexable stream with a stream identifier, or a pipe from a trusted process; see *pex*(4) and *stream*(4).

Privedit applies to the *privs* file the modifications given in the *changes* file. Only the part of the authorization tree rooted at the given *node* may be changed. The form of *changes* is described in *privs*(5). The changes are echoed and confirmation is requested. (*Privedit*, like any other *command*, is a conventional token defined by the *privs* file; it is not built in.)

Priv clears the environment to prevent hidden corruption by untrusted processes. For the same reason it asks confirmation of the argument list. What you see is what it will do.

The real work of *priv* is done by *privserv*(8). *Priv* communicates with *privserv* via a pipe that the latter mounts on `/cs/priv`.

FILES

`/etc/privs`
`/cs/priv`

SEE ALSO

privs(5), *privserv*(8), *session*(1)

DIAGNOSTICS

If a *command* is performed, *priv* returns the result of the last constituent action; see *privs*(5).

BUGS

Trailing null *args* are deleted.
The standard input and standard error cannot freely be redirected.
It is possible for a password to be demanded twice. This would be mitigated if requests were assessed in decreasing order of specificity instead of table order.

NAME

redmail, blackmail – multilevel mail

SYNOPSIS

`redmail`

`blackmail` *maildir*

DESCRIPTION

These commands arrange for the delivery and receipt of mail with varying security labels.

Redmail simulates *mail*(1) for reading multilevel mail.

To receive timely notification of multilevel mail, set the shell variable `MAIL` to *maildir*`/FLAG`.

Blackmail delivers multilevel mail to the private mail directory *maildir*. To arrange for incoming mail to be handled by *blackmail*, provide a directory *maildir* (conventionally `.mail` in your home directory) and edit a single line like

```
Pipe to blackmail $HOME/.mail
```

into your regular mailbox, `/usr/spool/mail/`*logname*. Thereafter a separate *blackmail* process runs for each letter. Letters receive the security label of their source.

FILES

`$HOME/.mail/*`

SEE ALSO

mail(1)

NAME

session, drop, runlow – substitute labels temporarily

SYNOPSIS

`session` [*option* ...]

`priv session` [*option* ...]

`runlow` *command*

`drop` [`-l` *label*] [*command-arg* ...]

DESCRIPTION

Session sets a temporary security label for the duration of one command. The ceiling is raised sufficiently to cover the requested label, up to the authorization recorded for the current login name. If no *command-args* are given, the command is taken to be a shell: *sh*(1) above the system floor, or *nosh*(8) below. With *command-args,* the specified command is run; there is no shell-like path search.

If the current ceiling does not dominate the new ceiling, or the the new process label is below the system floor and does not dominate the current label *session* must be invoked through *priv*(1).

The options are

`-l` *label*
: Set the process label and the label of the standard input to the given value, specified as in *atolab*; see *labtoa*(3). If the value does not dominate the current process label, clear the environment and pass no arguments to the invoked command. If *label* is missing, it is taken to be the system floor.

`-C` *label*
: Set the process ceiling at or above the given value. If *label* is missing, it is taken to be the process label.

`-u` *user* The password for *user* will be demanded. The fact that the password has been presented will be recorded in the stream identifier (see *stream*(4)) of the standard input. For the duration of the session, further queries for that password will succeed automatically. If *user* is missing, it is taken to be the current login name.

`-x` Replace current session instead of suspending it for the duration of the new session (like `exec` in *sh*(1)).

`-c` *command-arg ...*
: Instead of a shell, run the given command with the given arguments. This option must come last.

To change labels, the input source must come over a trustable channel, in particular neither from an untrusted computer nor from a terminal into which untrusted code has been downloaded. The request may require confirmation to assure that no software has tampered with it; answer `y` for yes. Confirmation and password inquiries happen under cover of *pex*(4). In a *mux*(9.1) window, this gives a visible indication; a missing indication is a sign of spoofing.

Runlow runs a command, starting the label at bottom, somewhat like `session -l 0`, but without changing the label of the standard input. The executable file is located according to environment variable `$PATH` as in *sh*(1). The command receives empty argument and environment lists, but inherits open file descriptors; only descriptors 0-3 are allowed. The process label will immediately rise to dominate that of the executable file.

Drop sets the process ceiling to *label* (by default to the process label) for the running of one *command* with the given *arguments.* If no *command* is given, `/bin/sh` is run.

The current process label, process licenses, terminal label, and environment are preserved.

EXAMPLES

`priv session -C ffff...`
: Change ceiling to the maximum authorized for the current user.

```
priv session -l 0
cd /usr/src
```

Enter a bottom-label interactive terminal subsession. Get out of the black-hole directory that *priv*(1) leaves you in.

```
runlow /bin/sh    # not useful
```

An attempt to fool the system into giving a bottom-label interactive shell. When the shell reads from standard input, its label will revert to that of the current session.

```
drop ls -l *
drop pwd
```

Prevent the process label from rising to cover the labels of files in the directories examined by *ls* or *pwd*. (If the label did rise, the output could not get to the terminal.)

FILES

```
/dev/log/sessionlog
/etc/pwfile
/etc/floor
/bin/sh
/etc/nosh
```

SEE ALSO

sh(1), *getflab*(2), *getplab*(2), *exec*(2), *pwfile*(5), *login*(8), *nosh*(8), *pwserv*(8)

DIAGNOSTICS

‘Sorry’, instead of asking for a password: untrusted channel.

NAME

setlab, downgrade, setpriv – set security label on files

SYNOPSIS

`setlab` [*option ...*] *label file ...*]

`priv downgrade` [`-v`] *delta file ...*

`priv setpriv` *cap lic file ...*

DESCRIPTION

Setlab sets the security label on the named *files*, or on the standard input if no files are named. The *label* is a single argument in the style accepted by *atolab*; see *labtoa*(3). The options are

`-a` Add *label* to the current file label (*new=old*|*label*).

`-s` Subtract *label* from the current file label (*new=old&~label*).

`-p` Set privileges (capabilities and licenses) only.

`-v` Print a blow-by-blow account on standard error file.

The process must be able to open the file, either for reading or writing. One or more licenses (see *getplab*(2)) are needed in some instances:

`T_EXTERN` to downgrade (new label does not dominate old)

`T_SETPRIV` if either the old or the new label has nonzero privilege bits

`T_NOCHK` if the old label has flag `L_NO` (also need `T_EXTERN` to change away from `L_NO`).

Downgrade uses *setlab* to clear the label bits designated by *delta*. It is a conventional request defined in the privilege file, *privs*(5), which checks that the user has authority over the specified label bits and supplies the necessary privilege to *setlab.*

Setpriv is a conventional interface to *setlab* for changing file capabilities and licenses.

EXAMPLES

`setlab ffff... file`
Give the file a top label.

`setlab -a F file`
Freeze a file label to keep writes from raising the lattice value.

`lmask x setlab -s 03 file`
Downgrade a security label using a privileged *nosh*(8) session. The *downgrade* priv request is preferred.

`priv downgrade 03 file`
Same, using obtaining the necessary authorization and privilege from *priv*(1).

`priv setpriv - n file`
Give the file a license, but no capabilities. This is a conventional trick to make the file immutable until its privileges are turned off again. The lattice value of the label is bottom (all zero).

DIAGNOSTICS

'Locking file for vetting'. As a matter of policy, *setlab* refuses to assign arbitrary privileges to a previously unprivileged ('untrusted') file. Instead it marks the file immutable as in the last example. The file may then be examined at leisure to assess whether its contents are indeed trustable before privileges are finally assigned.

SEE ALSO

getflab(2), *getlab*(1), *priv*(1)

BUGS

The strings `-a` and `-p` happen to be legitimate, if unusual, labels. They will always be understood as option flags.

NAME

stat – file statistics and labels

SYNOPSIS

`stat` *file* ...

DESCRIPTION

Stat places facts about the named *files* on the standard output. Successive output lines show

The file name.

Inode number, mode, link count, owner, group, and size displayed like output from *ls*(1) with options `lidL`. For device files, the size is replaced by major and minor device numbers separated by a comma.

The major and minor device numbers of this inode's file system and the file mode in octal.

Modification, access, and change times, each on a separate line.

The security label (of the destination, if a symbolic link) is given in the style of *labtoa*(3).

If the file can be opened and corresponding data differ for the opened file, similar information for the opened file follows.

If the file is a symbolic link, the link destination is given, marked by `->`.

Stat has nocheck capability; a superuser with nocheck license can use it to examine any file.

EXAMPLES

```
/dev/tty:
	0 crwxrwxrwx 0 root 0 0,0
	255,255 020777
	Jun 22 22:52:30 1988
	Jun 22 22:52:30 1988
	Jun 22 22:52:30 1988
	------ ------CY 0000 0000 ...
	974  rw-rw-r-- 0 reeds other 0
	255,255 0100664
	------ ------R  0000 0000 ...
/usr/spool/man:
	9926 lrwxrwxrwx 1 doug bin 23
	7,66 0120777
	Oct 17 21:21:21 1987
	Jun 22 22:52:14 1988
	Oct 17 21:21:21 1987
	------ ------C  0000 0000 ...
	-> /n/bowell/usr/spool/man
```

DIAGNOSTICS

Diagnostics appear on the standard output.

SEE ALSO

stat(2), *ls*(1), *getlab*(1)

NAME

intro, fmount, getuid, signal, stat, wait – changes to manual

SYNOPSIS

```
#include <sys/label.h>

int fmount5(type, fildes, name, flag, ceiling)
char *name;
struct label *ceiling;
```

DESCRIPTION

This section covers small changes in the named manual pages for IX relative to v10.

intro

New error returns:

38 `ELAB` Security label violation
: An action which would, if completed, break security rules; see *getplab*(2).

39 `ENOSYS` No such system call
: An attempt to execute a nonexistent or unsupported system call.

40 `ENLAB` Out of security labels
: A system table for security labels is full: a trouble similar to `ENFILE`.

41 `EPRIV` Insufficient privilege
: An attempt was made to execute a priviledged system call, or exercise a privileged feature of a regular system call.

fmount

Fmount5 mounts a file system as does *fmount*(2) and, on regular (type 0) or network (type 4) file systems, imposes a specified *ceiling* label. No file in the file system can be accessed unless the label of the file is dominated by the file system ceiling. Moreover, in determining capabilities during *exec*(2), capability and license bits in the file label are masked by corresponding bits in the file system ceiling. The default ceiling is `L_YES` on regular and `L_NO` on network file systems. Default capabilities and licenses are all zero.

getuid

Whenever one of the system calls *setuid*, *setgid*, *setruid*, or *setlogname* requires superuser status, it also requires capability `T_UAREA`.

Setpgrp can set the process group only to the current process id unless the process has capability `T_UAREA`.

signal

Security label violations by *write*(2) result in `SIGPIPE`. Other security label violations result in `SIGLAB`, which is ignored if not caught.

stat

New modes. These are indicated in *ls*(1) by `a` and `b` appended to the usual mode field.

`S_IAPPEND`
: Append-only file.

`S_IBLIND`
: Blind directory. A blind directory cannot be read and is immune to security label checks on search; files can be removed from it only by their owners.

wait

If the security label of the waiting process does not dominate that of the exiting process, then nonzero termination status or exit code is reported simply as `SIGTERM`.

NAME

execl, execv, execle, execve, execlp, execvp, exect, environ – execute a file

SYNOPSIS

```
int execl(name, arg0, arg1, ..., argn, (char *)0)
char *name, *arg0, *arg1, ..., *argn;

int execv(name, argv)
char *name, *argv[];

int execle(name, arg0, arg1, ..., argn, (char *)0, envp)
char *name, *arg0, *arg1, ..., *argn, *envp[];

int execve(name, argv, envp)
char *name, *argv[], *envp[];

int execlp(name, arg0, arg1, ..., argn, (char *)0)
char *name, *arg0, *arg1, ..., *argn;

int execvp(name, argv)
char *name, *argv[];

int exect(name, argv, envp)
char *name, *argv[], *envp[];

extern char **environ;
```

DESCRIPTION

Exec in all its forms overlays the calling process with the named file, then transfers to the entry point of the image of the file. There can be no return from a successful *exec*; the calling image is lost.

Files remain open across *exec* unless explicit arrangement has been made; see *ioctl*(2). Signals that are caught (see *signal*(2)) are reset to their default values. Other signals' behavior is unchanged.

Each user has a *real* userid and groupid and an *effective* userid and groupid. The real userid (groupid) identifies the person using the system; the effective userid (groupid) determines access privileges. *Exec* changes the effective userid and groupid to the owner of the executed file if the file has the set-userid or set-groupid modes. The real userid is not affected.

The security label (see *getflab*(2)) of the process is set as follows. If any arguments or environment parameters are present, or if and file descriptor numbers greater than 3 are in use, the lattice value of the process label is ascribed to them, otherwise lattice bottom. This value is ORed with the lattice value of the executed file to obtain the new lattice value for the process. If the new lattice value does not dominate the old, the permission mask (see *umask*(2)) is set to 022.

Process licenses persist. In the simplest case, the process obtains from the file the capabilities for which the process has licenses; see *getplab*(2). The detailed computation for process capabilities is: Nominal capabilities are determined by ANDing the file capabilities with the capabilities in the file system ceiling (see *mount*(2)) and then ORing with built-in minima. Nominal licenses are determined by ANDing the file licenses with the licenses in the file system ceiling and with built-in maxima. Process capabilities are set by ORing the process licenses with the nominal licenses, then ANDing with the nominal capabilities.

The builtin minimum file capabilities are all 0. The builtin maximum file licenses for `T_SETPRIV` and `T_LOG` are 0; the rest are 1.

The *name* argument is a pointer to the name of the file to be executed. If the first two bytes of that file are the ASCII characters `#!`, then the first line of the file is taken to be ASCII and determines the name of the program to execute. The first nonblank string following `#!` in that line is substituted for *name*. Any second string, separated from the first by blanks or tabs, is inserted between the first two arguments (arguments 0 and 1) passed to the invoked file.

The argument pointers *arg0, arg1*, ... or the pointers in *argv* address null-terminated strings. Conventionally argument 0 is the name of the file.

Execl is useful when a known file with known arguments is being called; the arguments to *execl* are the character strings constituting the file and the arguments.

Execv is useful when the number of arguments is unknown in advance; the arguments to *execv* are the name of the file to be executed and a vector of strings containing the arguments. The last argument string must be followed by a 0 pointer.

When a C program is executed, it is called as follows:

```
main(argc, argv, envp)
int argc;
char **argv, **envp;
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. Conventionally *argc* is at least 1 and `argv[0]` points to the name of the file.

Argv is directly usable in another *execv* because `argv[argc]==0`.

Envp is a pointer to an array of strings that constitute the *environment* of the process. Each string conventionally consists of a name, an `=`, and a null-terminated value; or a name, a pair of parentheses `()`, a value bracketed by `{` and `}`, and a null character. The array of pointers is terminated by a null pointer. The shell *sh*(1) passes an environment entry for each global shell variable defined when the program is called. See *environ*(5) for some conventionally used names.

The C run-time start-off routine places a copy of *envp* in the global cell *environ*, which is used by *execv* and *execl* to pass the environment to any subprograms executed by the current program. The *exec* routines use lower-level routines as follows to pass an environment explicitly:

```
execve(file, argv, environ);
execle(file, arg0, arg1, . . . , argn, (char *)0, environ);
```

Execlp and *execvp* are called with the same arguments as *execl* and *execv*, but duplicate the shell's actions in searching for an executable file in a list of directories given in the `PATH` environment variable.

Exect is the same as *execve*, except it arranges for a stop to occur on the first instruction of the new core image for the benefit of tracers, see *proc*(4).

FILES

`/bin/sh` shell, invoked if command file found by *execlp* or *execvp*

SEE ALSO

fork(2), *environ*(5)

DIAGNOSTICS

`E2BIG`, `EACCES`, `EFAULT`, `EIO`, `ELAB`, `ELOOP`, `ENOENT`, `ENOEXEC`, `ENOMEM`, `ENOTDIR`, `ENXIO`, `EROFS`, `ETXTBSY`

BUGS

If *execvp* is called to execute a file that turns out to be a shell command file, and if it is impossible to execute the shell, some of the values in *argv* may be modified before return.
The path search of *execlp* and *execvp* does not extend to names substituted by `#!`.

NAME

getflab, fgetflab, setflab, fsetflab – get or set file security label and privilege

SYNOPSIS

```
#include <sys/label.h>

getflab(name, labp)
char *name;
struct label *labp;

fgetflab(fildes, labp)
struct label *labp;

setflab(name, labp)
char *name;
struct label *labp;

fsetflab(fildes, labp)
struct label *labp;
```

DESCRIPTION

Getflab copies the security label from the the named file into the structure pointed to by *labp*. *Fgetlab* copies the security label from an open file specified by file descriptor. The field `lb_junk` is always zero.

The structure of a security label as defined in `<sys/label.h>` is

```
#define LABSIZ         60
struct  labpriv {
        unsigned int   lp_junk : 16,/* poison level, see syslog(2) */
                       lp_flag : 2,
                       lp_fix : 2,/* fixity */
                       lp_t : 6,/* capabilities */
                       lp_u : 6;/* licenses */
};
struct  label {
        struct labpriv lb_priv;
        unsigned char  lb_bits[LABSIZ];
#       define lb_junk lb_priv.lp_junk
#       define lb_flag lb_priv.lp_flag
#       define lb_t    lb_priv.lp_t
#       define lb_u    lb_priv.lp_u
#       define lb_fix  lb_priv.lp_fix
};
                       /* codes in lb_flag */
#define L_YES          1
#define L_NO           2
#define L_BITS         3
                       /* codes in lb_fix */
#define F_LOOSE        0
#define F_FROZEN       1
#define F_RIGID        2
#define F_CONST        3
                       /* bits of lb_t and lb_u */
#define T_SETPRIV      001      /* may set file privilege */
#define T_SETLIC       002      /* may change process license */
#define T_NOCHK        004      /* exempt from label checking */
#define T_EXTERN       010      /* may introduce foreign data */
#define T_UAREA        020      /* may write in u area */
```

```
#define T_LOG          040      /* may execure syslog() call */
```

Three types of labels are distinguished by the `lb_flag` field:

`L_YES` The file can be read or modified without regard to label. Its inode data (see *stat*(2)) have permanent conventional values. *Null*(4), *log*(4), and *fd*(4) are labeled `L_YES`.

`L_NO` The the file and its inode cannot be read or written except by processes with capability `T_NOCHK`. A `L_NO` label may be changed by processes with capability `T_EXTERN`, unless prevented by `F_CONST` described below.

`L_BITS` The label has a 'lattice value', given by `lb_bits` and so called because the values form a mathematical lattice with bitwise AND as the meet operation and OR as the join.

Each process and each file has a label. Normally data may only flow 'up' the lattice. The destination of a read, write, inode query, or inode change must have a lattice value that dominates (bitwise) the lattice value of the source, unless the process concerned has capability `T_NOCHK`.

To assure upward flow, a *read*(2) or an inode query (e.g. *stat*(2)) normally causes the file label to be OR-ed into the process label. Similarly a *write*(2) or an inode change (as by *chmod*(2) or *link*(2)) causes the process label to be OR-ed into the file label. However such side-effect changes in a file or process label may happen only if the label is loose (see below) and the new label is dominated by the process ceiling; see *getplab*(2). Otherwise the system call terminates with error `ELAB`.

Security checks are independent of, and made prior to, the permission checks described in *access*(2). Super-user processes are subject to security checks.

Setflab replaces the security label of the named file with the contents of the structure pointed to by *labp*. *Fsetflab* replaces the security label of an open file specified by file descriptor. If the new label has flag `L_BITS`, the new lattice value must dominate the old one, dominate the process label, and be dominated by the process ceiling. If the new label has flag `L_NO`, the old label must be dominated by the process ceiling. Flag `L_YES` is an error. The field `lb_junk` is ignored.

The field `lb_t` contains 'capability' bits; `lb_u` contains corresponding 'license' bits; their meanings are described in *getplab*(2) and *exec*(2). The two fields together are known as 'privileges'. Any file that has nonzero privileges is called 'trusted' and cannot be changed, in contents or in inode, except by processes with capability `T_SETPRIV`.

Aside from considerations of trustedness, a label can be changed with more or less freedom according to its 'fixity', `lb_fix`:

`F_LOOSE` Any process can change the lattice value of a loose file label implicitly as a side effect as described above or (up to the process ceiling) explicitly with *setflab* or *fsetflab*. The file owner or the super-user can change the fixity.

`F_FROZEN` The lattice value of a frozen label cannot change. The fixity can be changed by the file owner or the super-user.

`F_RIGID` Only processes with capability `T_EXTERN` can change a rigid label; see *getplab*(2). The labels of external media, such as terminals, tapes or disks, are automatically rigid. A loose or frozen label on a stream (see *stream*(4)) can be changed to rigid. This facility allows filters, such as *mux*(9.1), to make pipes behave like external devices. The fixity of a rigid label cannot change.

`F_CONST` A constant label may not be changed. The labels of certain special files, such as `/dev/null` and `/dev/mem`, are automatically constant; no other labels may become constant.

SEE ALSO

getplab(2), *getlab*(1), *labLE*(3), *setlab*(8), *unsafe*(2), *signal*(2)

DIAGNOSTICS

EFAULT, EIO, ELAB, ELOOP, ENOENT, ENOTDIR

NAME

getplab, setplab – get or set process security label and privilege

SYNOPSIS

```
#include <sys/label.h>

getplab(labp, ceilp)
struct label *labp, *ceilp;

setplab(labp, ceilp)
struct label *labp, *ceilp;
```

DESCRIPTION

Getplab copies the security label and the ceiling label, usually simply called 'the ceiling', of the current process into the structures pointed to by *labp* and *ceilp*. No copy happens for a zero pointer. The structure and meaning of labels are described in *getflab*(2). The ceiling is a security lid; the process can only access files with labels dominated by the ceiling.

A process may have special security 'capabilities', in which case it is called 'trusted'. The capabilities are obtained from the file it is executing, usually as 'licensed' from its parent process; see *exec*(2). The capabilities and corresponding licenses are given by bits in the fields *labp*`->lb_t` and *labp*`->lb_u` respectively. The bits are defined by the masks

`T_SETPRIV` The process can change the privileges of files; see *getflab*(2).

`T_SETLIC` The process can increase its own licenses; see below.

`T_EXTERN` The process can bring new data sources into view by mounting file systems or setting labels of (open) special files; see *getflab*(2).

`T_NOCHK` Ordinary checks and changes of lattice values are not made when reading or writing files or inodes or when setting the process label.

`T_UAREA` The process can change certain information that may be accessed by descendent processes without label checks; see *setuid*(2) and *stream*(4).

`T_LOG` The process can change logging status; see *syslog*(2).

Setplab copies the structures pointed to by *labp* and *ceilp* into the process label and the ceiling label. Unless the process has capability `T_NOCHK`, the new lattice value of the process label must dominate the old and the old lattice value of the ceiling must dominate the new.

The new label flag must be `L_BITS`, and the lattice value of the new ceiling label must dominate the lattice value of the new process label.

Capabilites may not increase. Licenses may increase only if the process has capability `T_SETLIC`.

The fixity, `lb_fix`, of a process may be set only to `F_LOOSE` or `F_FROZEN`. In the latter case the process label can not change as a side effect of label checking.

The bits of the ceiling pointer are themselves labeled as if they were a minifile. When the ceiling is set by *setplab*, the minifile label is set to the old value of the process label, unless the process has capability `T_SETLIC`, in which the minifile label is set to bottom. When the ceiling is read by *getplab*, the minifile label is checked as if read by *read*(2).

DIAGNOSTICS

`EFAULT`, `ELAB`, `EPRIV`

If *getplab* cannot raise the process label to dominate the minifile label, the requested labels are filled in, with the ceiling being censored to flag `L_NO ,` and `ELAB` is returned.

SEE ALSO

getflab(2), *unsafe*(2), *exec*(2), *session*(1), *setlab*(8)

NAME

labmount – return file system ceiling label

SYNOPSIS

```
int labmount(fd, lp)
struct label *lp;
```

DESCRIPTION

If the file with descriptor *fd* resides in a file system, *labmount* copies the ceiling label of that file system into the place pointed to by *lp*. If the file does not reside in a file system, the ceiling is reported to be `L_YES`; see *getflab*(2).

SEE ALSO

fmount(2)

DIAGNOSTICS

`EBADF, EFAULT, EIO, ELOOP, ENOENT, ENOTDIR`

NAME

nochk – control security checking by file

SYNOPSIS

```
nochk(fd, onoff);
```

DESCRIPTION

Nochk modifies file security checks in processes that have capability T_NOCHK. If *onoff* is 1, file descriptor *fd* becomes exempt from security checks; this is the default state. If *onoff* is 0, the file descriptor will be checked as if the process did not have capability T_NOCHK.

The return value is the previous checking state.

SEE ALSO

getplab(2)

DIAGNOSTICS

EBADF

BUGS

It would have been wise to let 0 be the default state, but this would have required modifying standard utilities, such as *fsck*(8), which must be run with privilege T_NOCHK.

NAME

seek, tell, lseek, llseek – manipulate read/write pointer

SYNOPSIS

```
int seek(fildes, offset, whence)
long offset

long tell(fildes)

long lseek(fildes, offset, whence) long offset;

Long llseek(fildes, offset, whence)
Long offset;
```

DESCRIPTION

Seek sets the file pointer for the file associated with *fildes* as follows:

If *whence* is 0, the pointer is set to *offset* bytes.

If *whence* is 1, the pointer is set to its current location plus *offset*.

If *whence* is 2, the pointer is set to the size of the file plus *offset*.

Tell returns the value of the file pointer associated with *fildes.*

Lseek is equivalent to *seek* followed by *tell.*

Llseek is like *lseek*, but handles i.e. 64-bit, file pointers.

Seeking far beyond the end of a file, then writing, creates a gap or ‘hole,’ which occupies no physical space and reads as zeros.

File pointers have security labels separate from files. For security-label calculations, *seek* is understood to ‘write’ the pointer, *tell* to ‘read’ it. If *whence* is 0 on *seek*, the new value of the file pointer does not depend on the old value.

SEE ALSO

open(2), *fseek*(3)

DIAGNOSTICS

`EBADF`, `ESPIPE`

BUGS

Lseek doesn’t affect some special files.

NAME

syslog – security logging

SYNOPSIS

```
#include <sys/log.h>

int syslog(command, arg2, arg3)
```

DESCRIPTION

Syslog controls security logging. The *command* argument determines the meaning of the other arguments.

Logging is done by writing on special files, described in *log*(4). One of these files is the 'system log file' where the kernel records certain events automatically. Each process has an 'audit mask' that determines which events cause logging records; mask items are defined in `<sys/log.h>`; see *log*(5). Each file has a 'poison class' with value 0, 1, 2, or 3. The kernel has a table of four corresponding 'poison masks' and a global audit mask. When a system call mentions a file in a pathname, the poison mask corresponding to the file's poison level is ORed into the process audit mask; when a process executes a file, the global log mask is ORed into the process audit mask.

The forms of the several *syslog* commands follow. Arguments shown as `0` are ignored.

`syslog(LOGON,` *fd* `,` *x* `)`
: Turn logging on; nominate file descriptor *fd* as repository for log file with minor device number *x*. *Fd* must be open for writing. Logging will persist after *fd* is closed.

`syslog(LOGOFF, 0,` *x* `)`
: Turn logging off on minor device number *x*.

`syslog(LOGGET,` *n*`, 0)`
: Return the value of the *n*th poison mask; *n*=4 designates the global audit mask.

`syslog(LOGSET,` *n*`,` *x*`)`
: Set the *n*th poison mask to *x*.

`syslog(LOGFGET,` *fd*`, 0)`
: Return the poison level of the file associated with file descriptor *fd*, which may be open for reading or writing.

`syslog(LOGFSET,` *fd*`,` *x*`)`
: Set the poison level of the file associated with file descriptor *fd*, (which may be open for reading or writing) to *x*. The poison level is stored in field `di_label.lb_junk` of the file's inode; see *inode*(5).

`syslog(LOGPGET,` *pid*`, 0)`
: Return the audit mask of process *pid*.

`syslog(LOGPSET,` *pid*`,` *x*`)`
: Set the audit mask of process *pid* to *x*.

Syslog works only in processes with capability `T_LOG`; see *getplab*(2).

SEE ALSO

log(4), *log*(5), *syslog*(8)

DIAGNOSTICS

`EBADF, EFAULT, EINVAL, EIO`

NAME

unsafe – detect potential file security violations

SYNOPSIS

```
#include <sys/types.h>

unsafe(nfd, readfds, writefds)
fd_set *readfds, *writefds;
```

DESCRIPTION

Unsafe examines file descriptors 0 through *nfd–1* and sets the corresponding bits of the masks in *readfds* and *writefds* to indicate files that are not known to be safe (i.e. to satisfy security label rules) for reading or writing respectively. The bit masks are indexed and manipulated as described in *select*(2).

At the same time, if the process has capability `T_NOCHK` (see *getplab*(2)), all file descriptors indicated by ones among the first *nfd* bits of the previous values of *readfds* and *writefds* are marked safe to read or write respectively.

Potentially unsafe situations arise from changes in file label caused by this or other processes, changes in process label, and file opening.

To prevent unintended violations of security policy, programs with capability `NOCHK` must monitor label changes. For this purpose the process label may be frozen (see *getplab*(2)) to prevent unintended automatic label changes. `SIGLAB` may be used to detect changes in file labels (see *signal*(2)), and *unsafe* to pinpoint them.

DIAGNOSTICS

`EFAULT`

SEE ALSO

getflab(2), *getplab*(2), *signal*(2), *select*(2)

NAME

buildmap, transin, transout – translate labels between computers

SYNOPSIS

```
#include <cbit.h>
struct mapping *buildmap(int fd, char *file, char *me, char *pw, server);
transin(struct mapping *map, struct label *foreign,struct label *domestic);
transout(struct mapping *map, struct label *domestic, struct label *foreign);
```

DESCRIPTION

Buildmap and its partner (which may be another instance of *buildmap*) at the far end of the stream *fd* work out a mapping for translating labels. The label space at the near end is defined by *file* , which contains ASCII representations of *cbit*(3) structures. The identity (for authorization purposes) of the near end is *me*, using a secret password *pw*. Each end challenges the other, using its own password to compose a response, whose validity is checked by *verify ;* see *notary*(3); The ends need not know each other's passwords. *Server* is to simplify the protocol: one end should have server set to zero, the other end to one.

Transout and *transin* translate labels according to the formula determined by *buildmap*. Labels in transit are represented in the binary form of the source machine and are translated on receipt, so *transout*'s job is simpler. Both routines return 0 if translation is impossible or illegal, otherwise they perform the translation and return 1.

SEE ALSO

cbit(3), *notary*(3)

DIAGNOSTICS

These routines all return 0 on error.

NAME

cbit, cbitread, cbitlookup, cbitparse, cbitcert – security category manipulation

SYNOPSIS

```
#include <cbit.h>

struct cbit *cbitparse(char **fields, struct cbit *cb);

struct cbit *cbitread(char *file);

struct cbit *cbitlookup(char *name, struct cbit *cb);

char *cbitcert(struct cbit *p);
```

DESCRIPTION

These functions manipulate certificates entitling computers to handle compartmented security categories. Each security compartment is represented by a structure of form:

```
struct cbit {
        char *name;             official name of category
        int floor;              default value (only bottom bit used)
        char *owner;            public name of issuing authority
        char *nickname;         our version of category name
        int bitslot;            where we store it
        char *exerciser;        who we are
        char *certificate;      owner's signature
}
```

which describes the meaning of the *bitslot*-th bit in a computer's label space. By convention, the lines of the file `/etc/cbits` contain (in ASCII colon-separated form) the compartments currently held on the local computer.

Cbitparse fills in and returns a cbit in the obvious way from a vector of seven strings. If the second argument is zero *cbitparse* allots a new structure using *malloc*(3).

Cbitread reads and parses an ASCII file of cbits, returning an array of filled in structures. The last entry in the array is a dummy; it is signalled by having a zero value of *name*.

Cbitlookup, when fed a category name and an array of cbits (such as returned by *cbitread*), returns a pointer to the unique entry whose category name is *name*, or returns zero.

Cbitcert composes a certificate granting *exerciser* the right to hold files with the given security category. The output of *cbitcert* depends only on *name*, *floor*, *owner*, and *exerciser*. The output must be signed by *owner* with *xs* (see *notary*(3)) to produce the checksum value in *certificate*. Third parties may check validity of a cbit by calling

```
verify(p->exerciser, p->certificate, cbitcert(p), strlen(cbitcert(p)))
```

FILES

```
/etc/cbits
```

SEE ALSO

notary(3).

DIAGNOSTICS

These routines all return 0 on error.

NAME

getstsrc, setstsrc – read and write a stream identifier

SYNOPSIS

```
char *getstsrc(fd)
setstsrc(fd, name)
char *name;
```

DESCRIPTION

Setstsrc attaches a descriptive string to the indicated stream, and *getstsrc* returns a pointer to a static buffer containing the string. The string persists until final close of the stream. When a stream is first opened the string is trivial.

Setstsrc requires capability `T_EXTERN`; see *getplab*(2). The string conventionally names the off-machine source of the stream. Since only trusted processes may modify it, it may be relied on for security calculations.

Getstsrc returns 0 on error, *setstsrc* returns –1 on error.

SEE ALSO

stream(4)

DIAGNOSTICS

`EPERM`, `ENOTTY`

BUGS

The return value of *getstsrc* points to static data whose content is overwritten by each call.

NAME

labelyes, labelno, labeltop, labelbot – label constants

SYNOPSIS

```
extern struct label labelyes;
extern struct label labelno;
extern struct label labeltop;
extern struct label labelbot;
```

DESCRIPTION

These objects are initialized as follows, where the coded values are as in *labtoa*(3).

`labelyes`	The universally permissive label, `Y`.
`labelno`	The universally denying label, `N`.
`labeltop`	The top lattice value, `ffff....`
`labelbot`	The bottom lattice value, `0000....`

NAME

labeq, lable, labmax, labmin – compare security labels

SYNOPSIS

```
#include <sys/label.h>

labEQ(x, y)
struct label *x, *y;

labLE(x, y)
struct label *x, *y;

struct label labMAX(x, y)
struct label *x, *y;

struct label labMIN(x, y)
struct label *x, *y;
```

DESCRIPTION

LabEQ returns 1 if *x* and *y* point to equal labels, otherwise 0. The result is 1 if and only if neither argument is 0, the flag fields are the same, and, when the flag fields are `L_BITS`, the lattice values are the same.

LabLE returns 1 if the security label pointed to by *x* compares less than or equal to the security label pointed to by *y*. An improper argument is treated as if it had flag `L_NO`. If one of the labels has flag `L_YES`, the result is 1; otherwise if one of the labels has flag `L_NO`, the result is 0; otherwise the the result is 1 if and only if the lattice value of *x* is bitwise less than or equal to the lattice value of *y*. (Inequalities involving `L_YES` and `L_NO` are not transitive.)

LabMAX and *labMIN* respectively return the maximum (bitwise OR) and minimum (bitwise AND) of lattice values of labels pointed to by *x* and *y*. An improper argument is treated as if it had flag `L_NO`. If one of the labels has flag `L_YES`, the result is the other label; otherwise if one of the labels has flag `L_NO`, the result has flag `L_NO`.

The privilege and frozen-label fields of the labels are disregarded by all of these functions.

SEE ALSO

getflab(2)

NAME

labtoa, atolab, atopriv, privtoa – security label conversion

SYNOPSIS

```
#include <sys/label.h>

char *labtoa(labp) struct label *labp;
struct label *atolab(string) char *string;

atopriv(string) char *string;

char *privtoa(n)
```

DESCRIPTION

Labtoa returns a pointer to a null-terminated ASCII string that represents the value of the security label pointed to by *labp*. The string has a form exemplified by

```
guxnlp guxnlpFY 0000 0000 ...
```

The characters of the first group `guxnlp` denote capabilities `T_LOG`, `T_UAREA`, `T_EXTERN`, `T_NOCHK`, `T_SETLIC`, and `T_SETPRIV` respectively. Characters of the second group denote corresponding licenses; see *getplab*(2). Missing capabilities or licenses are denoted by `-`.

The character shown as `F` denotes the fixity of the label. It may be a space (loose), `F` (frozen), `R` (rigid), or `C` (constant) The character shown as `Y` denotes the label's flag. It may be a space for a lattice label, `N` for `L_NO`, `Y` for `L_YES`, or `U` for the erroneous flag value 0.

Each group of four zeros may be any four lower case hex digits representing the value of two bytes of the lattice value. Repeating groups at the end of the string are denoted `...`.

Atolab inverts the process. The order of characters in, and length of, privilege strings are arbitrary, except that a nonempty license string must be preceded by a nonempty capability string. The order of characters from the set `YNUFRC` is arbitrary. Spaces must separate nonempty capability and license strings, and may be interspersed arbitrarily after the license string. A final `...` causes the last four hex digits to be repeated, provided the preceding label contains a multiple of four digits. A short or missing lattice value is padded with zeros.

Atopriv converts a string of characters from the set `guxnlp-` into privilege bits that may be stored in the `lb_t` or `lb_u` fields of a label structure. The order and number of characters are arbitrary.

Privtoa is inverse to *atopriv*.

SEE ALSO

getflab(2), *getplab*(2), *getlab*(1)

DIAGNOSTICS

Atolab returns 0 for unrecognizable input.

Atopriv returns the negative value `~(T_LOG|T_UAREA|T_EXTERN|T_NOCHK|T_SETLIC|T_SETPRIV)` for unrecognizable input.

BUGS

The value returned by *labtoa*, *atolab*, or *privtoa* points to a static buffer that is overwritten at each call.

NAME

xs, enroll, verify, reverify, keynotary – certification functions

SYNOPSIS

```
char *xs(char *key, char *buf, int n)
enroll(char *name, char *oldkey, char *newkey)
verify(char *name, char *xsum, char *buf, int n)
rverify(char *name, char *xsum, char *buf, int n)
keynotary(char *key1, char *key2)
```

DESCRIPTION

All these funcitons except *xs* must be linked with option `-lipc` of *ld*(1).

Xs composes a cryptographic checksum of the *n* characters starting at *buf*. The *key* argument points to an 8-character checksumming key. A pointer is returned to a null-terminated ASCII checksum.

Enroll registers a checksumming key for user *name* with *notary*(1), only one checksumming key per user name at a time. On first registry the *oldkey* argument is ignored. On subsequent registries, the *oldkey* argument must match the currently stored checksumming key. The new checksumming key is *newkey*; if *newkey* is trivial, *name* is deregistered.

Verify consults the notary oracle to check the validity of a checksum composed by *xs*. A non zero return value signifies that the checksum was calculated using the checksumming key registered with the notary oracle as belonging to user *name*. *Rverify* does what *verify* does, but leaves the connection to the oracle open until presented with a NULL value for *name*. Hence, subsequent calls to *rverify* should be quicker.

Keynotary is used to tell the notary daemon the key for its private encrypted data. *Key1* is the key the data is currently encrypted with; *key2* (if nonzero) is the key to use in the future. A file descriptor is returned, from which diagnosic information may be read.

SEE ALSO

notary(1), *ipc*(3)

DIAGNOSTICS

Verify and *enroll* return zero on failure, otherwise nonzero.

NAME

pex, unpex – obtain process-exclusive file access

SYNOPSIS

```
#include <sys/pex.h>

int pex(fd, seconds, pexbuf)
struct pexclude *pexbuf;

int unpex(fd, seconds)
```

DESCRIPTION

Pex tries, using the ioctl call, to obtain exclusive access to the file designated by file descriptor *fd*; see *pex*(4). If *pexbuf* is nonzero, facts about the other end of the pipe are placed in the object *pexbuf* points to, as described in *pex*(4).

If *fd* refers to a stream, *pex* first empties the input and output queues, flushing if *seconds* is negative, and otherwise waiting up to the specified time interval for the queues to drain. If the queues do not drain, an error results.

Unpex uses to try to reverse the effect of *pex*, again flushing or draining queues as specified by *seconds.*

On a pipe, *pex* or *unpex* succeeds only if the process at the other end answers with an FIOPX or FIONPX ioctl respectively. *Pex* and *unpex* should not be used to answer.

SEE ALSO

pex(4)

DIAGNOSTICS

Pex returns –1 on failure, 0 on success, and 1 for a half-pexed pipe.

NAME

pwquery, pexpw – password services

SYNOPSIS

```
pwquery(fd, name, param)
char *name;
char *param;

char *pexpw(fd, prompt)
char *prompt;
```

DESCRIPTION

Pwquery calls upon the password server, *pwserv*(8) to cause a password to be demanded from file descriptor *fd* and checked against the password for the named user. It is loaded by option `-lipc` of *ld*(1).

Echoing is disabled during the transaction, and the server is persnickety about when to use an Atalla challenge/response dialogue and when to use *crypt*(3)-style passwords. A negative return value indicates a protocol error in reaching the server or that the server is not trusted. A zero return value indicates rejection of the password. A positive return value indicates approval of the password.

The argument *param* may be zero (for vanilla password service) or may point to a blank-separated list of one or more keywords. Currently only one keyword is understood:

`pex` Reject the password if the stream is unpexable.

Pexpw reads a password from the indicated file descriptor, after prompting with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most 8 characters. The dialogue is not attempted if it cannot be protected from eavesdropping by the process-exclusive mechanism of *pex*(4).

FILES

```
/cs/pw
/etc/pwserv
```

SEE ALSO

ipc(3), *getpass*(3), *pex*(4) *pwserv*(8)

BUGS

Pexpw returns a pointer to static memory that is overwritten at every call.

NAME

proc, stream – changes to manual

DESCRIPTION

This section covers small changes in the named manual pages for IX relative to v10.

proc

The groupid of all files in `/proc` is –1. No process can have this groupid.

stream

Changed *ioctl*(2) calls for streams:

`TIOCSPGRP`

The process group can be set only to the current process id unless the process has capability `T_UAREA`.

`FIORCVFD`

Deliver a structure pointed to by *param*:

```
struct passfd {
      int   fd;
      short uid;
      short gid;
      short nice;
      char  logname[8];
      char  cap;
};
```

The call blocks until there is something in the stream. If data is present, it returns `EIO`. If a file descriptor has been sent from the other end of the pipe by `FIOSNDFD`, `FIORCVFD` fills in the user and group ID of the sending process, its niceness (see *nice*(2)), its login name, its capabilities in the form of the field `lb_t` (see *getflab*(2)), and a file descriptor for the file being sent; the file is now open in the receiving process.

New *ioctl* calls:

`FIOGSRC`

Copy the stream identifier to the `SSRCSIZ`-byte string pointed to by *param.*

`FIOSSRC`

Copy the `SSRCSIZ`-byte string pointed to by *param* into the stream identifier. Capability `T_EXTERN` is required; see *getplab*(2). A newly created stream has an empty stream identifier. It is customary to set the stream identifier on network connections to identify the source. Successful password demands may also be recorded in the stream identifier for the benefit of *pwserv*(8).

SEE ALSO

session(1), *src*(5)

NAME

log – security log file

SYNOPSIS

```
#include <sys/log.h>
```

DESCRIPTION

The special files `/dev/log/log00` through `/dev/log/log15` refer to ‘repository’ files nominated by *syslog*(2).

The kernel automatically records selected events on the ‘system log file’ `/dev/log/log00` in the form described in *log*(5).

Any process with write access may write on a log file; no process has read access. Each write places in the repository file a *log*(5) record with `code` = `LOG_USER`. The data written are truncated to `LOGLEN` bytes and placed in the `body` field. When logging is not turned on, a log file acts like a write-only `/dev/null`.

FILES

`/dev/log/*`

SEE ALSO

syslog(2), *log*(5), *syslog*(8)

NAME

pex – ioctl requests for process-exclusive access

SYNOPSIS

```
#include <sys/pex.h>

ioctl(fildes, FIOPX, p)
struct pexclude *p;

ioctl(fildes, FIONPX, p)
struct pexclude *p;

ioctl(fildes, FIOQX, p)
struct pexclude *p;

ioctl(fildes, FIOAPX, p)
struct pexclude *p;

ioctl(fildes, FIOANPX, p)
struct pexclude *p;
```

DESCRIPTION

These *ioctl*(2) requests provide and check temporary exclusive access to an input/output source. `FIOPX` marks as 'pexed' the file or pipe end referred to by *fildes*. On a pexed file *read*, *write*(2), and most forms of *ioctl* work only in the pexing process. Moreover, these operations do not work in any process on a half-pexed pipe (a pipe with exactly one pexed end). The mark remains until the pexing process requests `FIONPX` or closes all file descriptors that refer to the file.

When *fildes* refers to a stream, `FIOPX` and `FIONPX` require the stream's input and output queues to be empty; *pex*(3) gives a method for emptying them. When *fildes* refers to a pipe, the far end of which is unpexed, `FIOPX` waits, with timeout, for an answering `FIOPX` or `FIONPX` at the far end. `FIONPX` waits similarly when the far end is pexed. Either request returns 1 when it leaves a pipe with exactly one end pexed. A pipe must cycle through the fully unpexed state between fully pexed states; from the time one end becomes unpexed until the far end does too, `FIOPX` on the unpexed end will return error `ECONC`.

If argument *p* is nonzero, the structure it points to is filled in with information about the pexedness of the file and about the process at the far end of a pexed pipe. The format, defined in `<sys/filio.h>` is:

```
struct pexclude {
      int oldnear;      /* FIOPX or FIONPX: state at begining of call */
      int newnear;      /* FIOPX or FIONPX: state at end of call */
      int farpid; /* -1 if not pipe, 0 if not pexed, else process id */
      int farcap; /* if farpid>0, capabilities */
      int faruid; /* if farpid>0, user id */
};
```

Capabilities are represented as in the `lb_t` field of a label; see *getflab*(2).

`FIOQX` obtains the information without affecting state.

Read, *write*, or *ioctl* calls that fail due to pexedness return error `ECONC`. The only *ioctl* requests that may succeed on a half-pexed pipe are `FIOCLEX`, `FIONCLEX`, `FIOPX`, `FIONPX`, and `FIOQX`. A half-pexed pipe is deemed ready by *select*(2).

`FIOANPX` and `FIOAPX` modify the response of open stream device files to `FIOPX` requests. They require `T_EXTERN` capability; see *getplab*(2). After `FIOANPX` all `FIOPX` requests on the special file return 1 and leave the device in an unusable state (as if the device driver were a process at the far end of a pipe, always responding `FIONPX`). The treatment is reversed with `FIOAPX`. This mechanism allows a terminal to be denounced to the kernel as being attached to an untrusted remote computer that cannot guarantee the exclusivity asked by `FIOPX`.

EXAMPLES

A program collecting a password wishes to exclude other programs from the dialogue. The following code

does the trick. (When the dialogue passes through *mux*(9.1) or *con*(1), downstream stages of the path to the terminal can be assumed to be similarly pexed, provided FIOPX succeeds.)

```
#define ok(p) (p->farpid==-1 || p->farpid>0 && p->farcap!=0)
struct pexclude x;
if(ioctl(fd, FIOPX, &x) == 0 && ok(&x)) {
      static char buf[9];
      write(fd, promptstr, strlen(promptstr));
      read(fd, buf, 8);
      s = buf;
} else
      s = 0;
ioctl(fd, x.oldnear, 0);      /* restore state */
```

An intervening trusted program, with a policy of recognizing exclusive access only for trusted processes, may cooperate with

```
n = read(fd, buf, BUFSIZE);
if(n == -1 && errno == ECONC) {
      if(ioctl(fd, FIOPX, &pexcode)!=0 || pexcode.farcap==0)
            ioctl(fd, FIONPX, 0);
      } else  /* improper pexing */
```

SEE ALSO

ioctl(2), *pipe*(2), *stream*(4), *pex*(3)

DIAGNOSTICS

EBADF, ECONC, EFAULT, EIO, ENOTTY (FIOAPX and FIOANPX)
ECONC for forbidden IO calls in other processes.
EBUSY for an undrained queue.

NAME

filsys, fstab, passwd – changes to manual

DESCRIPTION

This section covers small changes in the named manual pages for IX relative to v10.

filsys

A disk inode in a regular file system contains an extra field for the file's security label.

```
#include <sys/label.h>
struct label labeldi;
```

fstab

The table of normally mounted file systems, `/etc/fstab`, contains an extra field for the file system ceiling; see *fmount*(2).

passwd

In addition to the usual password file, `/etc/passwd`, there is a highly secret file `/etc/pwfile`, which is used by *pwserv*(8) to authorize clearances. Each content line contains the following fields, separated by colons:

name
encrypted password
SNK key
process license (unused)
clearance (maximum ceiling)

The license and label fields are in the form understood by *labtoa*(3); thus the label field may contain white space. Lines with fewer than five fields are ignored.

The name field contains a user name for option `-u` of `/bin/session`. It is customary, but not necessary, for names in *pwfile* also to be registered in *passwd*(5).

The SNK field gives a 24-digit octal key for a Secure Net Key (or Atalla) challenge box.

The label field gives the maximum permissible label for option `-l`, and the ceiling label otherwise.

NAME

log – format of security logging records

SYNOPSIS

```
#include <sys/log.h>
```

DESCRIPTION

The structure of system log file records as declared in `<sys/log.h>` is

```
struct logbuf {
        short  len;          /* total length of whole record */
        short  pid;          /* process id */
        long   slug;         /* transaction number */
        char   code;         /* kind of record */
        char   mode;         /* sub-kind */
        char   colon;        /* ':', aids sync */
        char   body[LOGLEN];
};
```

The `code` field identifies the kind of record; for legal values see the include file. In kernel records the `mode` field identifies where in the kernel the logging record originated, for user records it contains the minor device number of the `/dev/log/log`*xx* file used to create the record.

The `body` field contains the logging record proper; its actual length is determined from the `len` field. In kernel records the `body` is a sequence of values, each prefixed by one or more format bytes according to the following list. Multibyte numbers are represented low byte first.

`s` Next two bytes are a byte count for following string.

`$` Next one byte is a byte count for following string, which is typically a file component name.

`C` Next byte is a byte count for following string, which is the command name.

`j` Next value is a security label: two bytes of `lb_priv` followed by two bytes of index into the kernel's shared label table for the lattice value of the label; see *getflab*(2).

`J` Next value is a security label: two bytes of `lb_priv` followed by two bytes of index into the kernel's shared label table for the lattice value of the label, followed by 60 bytes of bits of the lattice value of the label.

n Next *n* bytes (n=*1,2,3,4)* represent a number.

`I` Next bytes name an inode: two bytes of device followed by two bytes of inumber.

`E` The current system call suffered an `ELAB` error.

`e` Next byte is an *errno* code; see *intro*(2).

The various bits of the log mask (see *syslog*(2) are named `LN`, `LS`, `LU`, `LI`, `LD`, `LP`, `LL`, `LA`, `LX`, `LE`, `LT`, with the same meanings as the corresponding key letters defined in *syslog*(8).

FILES

`/dev/log`

SEE ALSO

syslog(2), *log*(4), *syslog*(8)

BUGS

The various kinds of kernel logging records are understandable only by reading the kernel source code.
It takes 7 bytes, not 4, to name an inode.

NAME

privs – privilege file

DESCRIPTION

The file `/etc/privs` expresses the rules whereby *priv*(1) grants privilege. It consists of a list of statements, each terminated by a semicolon. One or more comments, each extending from # to newline, may precede each statement.

Rights

Rights are defined thus:

`DEFINE` *rights-list* `;`

Each right in the comma-separated rights-list has a name, and optionally a parenthesized parameter type. The types are

`LAB` Label, ordered by lattice value.

`RE` Regular expression ordered by language inclusion. Regular expressions are in the form of *regexp*(3), with enclosing `^(` and `)$` understood.

`PRIV` Set of privileges in *atopriv* form, ordered by inclusion; see *labtoa*(3).

Examples:

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

Rights are identifiers used solely by *priv*; they have no other manifestation in the system. In the example, the `ceiling` right involves label comparisons, but has no necessary connection to process ceilings. The name could be changed globally to, say, `floor` without affecting the interpretation of `/etc/privs`.

Authorization

Authorization is expressed by a tree. Nodes of the authorization tree are named, like files in the file system, by full pathnames starting from the root, `/`. Associated with each node are statements to grant rights, and statements to admit access to the node. Rights are monotone in the tree: the rights at a node must be a subset of the rights at its parent. Access to a node implies access to its children.

Right-granting statements have the form

`RIGHTS` *nodename rights-list* `;`

A *rights-list* is as in a rights definition, but with explicit values for parameters. White space or one of the metacharacters `;,()` may be included in a value by placing double quotes around it. Examples:

```
RIGHTS /admin            priv(upxnl), ceiling(ffff...);
RIGHTS /admin/security   priv(p), ceiling("ffff ...");
RIGHTS /admin/operations priv(xn)
```

Access statements have the form

`ACCESS` *nodename pred-list* `;`

Access to the named node is granted when the comma-separated *pred-list* is nonempty and all the predicates in the list are satisfied. A node may have more than one `ACCESS` statement. Legal predicates are

`ID(` *lognames* `)`
A regular expression for login names that have access to this node.

`PW(` *name ...* `)`
The password associated with one of the *names* in *pwfile*(5) must be presented.

`SRC(` *source* `)`
A regular expression for the stream identifier of the standard input.

Rules

Rules give patterns for requests and show the prerequisite rights for and the actions to carry out each request:

`REQUEST(` *arguments* `) NEEDS` *rights* `DOES` *actions* `;`

The request part shows *arguments* supplied to *priv*(1); normally the arguments spell out the prefix of a UNIX command. The `NEEDS` part tells what rights are needed to perform the request. The rights are as in a rights statement, with substituted parameters; see 'Parameter values'.

If the process has access to a node that grants the needed rights (with the parameter in each grant dominating the parameter of the corresponding need), then the *actions* for the request are performed. Otherwise the request is denied. Legal actions are

`PRIV(gunxlp)`
Set one or more process licenses, abbreviated as in *labtoa*(3).

`EXEC(` *args* `)`
Execute a program given by the *args.* Members of the *args* list are separated by white space and may specify substitutions; see 'Parameter values'. `EXEC` does not do a *sh*(1)-like `$PATH` search.

`DAEMON(` *args* `)`
Same as `EXEC`, but do not wait for the command to complete.

`CEILING(` *label* `)`
Set the process ceiling.

`PRIVEDIT(` *node file* `)`
Read editing commands from the named *file*. Only the subtree at *node* is editable; nodes closer to the root cannot be touched.

`ANYSRC`
Skip the normal check for a trusted source; see *priv*(1).

The order in which nodes of the authorization tree are visited in evaluating a `NEEDS` clause is undefined, however at each node the predicates of the request are evaluated in order. The actions of a granted request are also performed in order, with effects such as privilege settings persisting until the end of the *priv* command or until overridden by a later action.

Parameter values

Parameter values appear in members of `NEEDS` and `DOES` lists. A value may be surrounded by double quotes, in which case the value may contain white space or one of the metacharacters `,;()`. A value may contain substitution marks, `$0`, `$1`, ... Each such mark is replaced from the *priv* invocation, `$0` standing for the match to the first *argument* of the `REQUEST` and so on. If a star is appended to the mark (e.g. `$0*`, `$1*`), the argument and all following ones are copied into the parameter list. Nothing can follow a star mark in a parameter.

Editing

Statements of the above forms may be used with action `PRIVEDIT` to augment a *privs* file. Further types of statements exist for editing only:

`RMDEFINE` *rights-list* `;`
Remove all occurences of the listed rights from the file.

`RMACCESS` *nodename pred-list* `;`
`RMRIGHTS` *nodename rights-list* `;`
Remove the given access list or the given rights from the named node. If the list is empty, remove all access lists or rights.

`RMREQUEST(` *arguments* `);`
Remove the `REQUEST` with identical *arguments.*

`RMNODE` *path-list* `;`
Remove the listed subtrees.

`DEFINE`, `RMDEFINE`, `REQUEST`, and `RMREQUEST` are understood to modify the root.

EXAMPLES

```
REQUEST(session -l)
      NEEDS ceiling($2)
      DOES PRIV(nx) EXEC(/bin/session -l $2);
REQUEST(/etc/downgrade -l)
      NEEDS downgrade($2)
      DOES PRIV(nx) EXEC($*);
```

FILES

`/etc/privs`

SEE ALSO

priv(1), *privserv*(8)

BUGS

There is no way to quote a newline or an initial double quote in parameters.

If an `ACCESS` or `RMACCESS` statement contains duplicate predicates, `RMACCESS` may remove an unintended list.

NAME

src – form of a stream identifier

DESCRIPTION

Stream identifiers, defined in *stream*(4), are conventionally set by *init*(8) and *dkmgr*(8) to designate the source of the login stream. A datakit source begins with `dk!` followed by a dial string.

Session(1) may append to the stream identifier of the standard input a colon and a name, which is understood by *pwserv*(8) as an assertion that the agent on that stream knows the password associated with that name, which obviates further demands for that password.

EXAMPLES

```
dk!201/mu/attbl:doug
```

SEE ALSO

getstsrc(3)

NAME

apx – mark an open stream device trusted

SYNOPSIS

`/etc/apx` [*arg*]

DESCRIPTION

By default, a freshly opened stream device has the `APX` bit cleared: it will reject all pex requests. If invoked without an argument, *apx* will set the APX bit on its standard input (by calling the `FIOAPX` control). If invoked without an argument the `APX` bit is cleared. *Apx* needs licence `T_EXTERN` to run. It is usually automatically invoked at login time, provided that the source identifier of the standard input of the login session is worthy.

FILES

`/etc/privs`

SEE ALSO

pex(4)

NAME

init, mount – changes to manual

DESCRIPTION

This section covers small changes in the named manual pages for IX relative to v10.

init

In single-user operation, *init* invokes *nosh*(8) with security label set at bottom and all capabilities; see *getplab*(2). At the end of single-user operation, *init* invokes *nosh* to run the startup script `/etc/rs.nosh`, again with bottom label and all capabilities.

In multiuser operation, *init* opens each terminal port with a security label set to the a ‘floor’ value, which is the label of the file `/etc/floor`.

mount

New option:

`-l` *label*

The *label*, specified as in *labtoa*(3), becomes the file system ceiling; see *fmount*(2).

NAME

cl, integrity – file system label check

SYNOPSIS

`/etc/cl` [*specfile* | *dir*] ...

`/etc/integrity` [*rootdir*]

DESCRIPTION

Cl examines file trees for correctness of labels. Each *specfile* argument names a file containing a description of the labels expected in a given subtree of a file system. Each line of a *specfile* has the form

```
filename uid,gid mode capabilities licenses label
```

User and group ids are specified in the style of *chown*(8). The mode is specified in the style of *chmod*(2); only the 07777 bits are significant. Capabilities and licenses are in the style of *atopriv*; see *labtoa*(3). The label is in the style of *atolab*, without capabilities or licenses.

The first valid line names the root of the tree in question. Subsequent lines name particular files in the tree. A report is made for each 'suspicious' file and for each particular file which does not match its description in *specfile*.

A suspicious file is a file that is not named in the *specfile* for which one of the following holds:

The label has flag `L_UNDEF` or `L_YES`.
The file is a special file the label flag is `L_NO`.
The file is not a special file the label flag is not `L_NO`.
The lattice value of the label is not dominated by the label in the first line of *specfile*.
The capability or license is not dominated by the corresponding value in the first line of *specfile*.

Each named directory argument *dir* is treated as if there were a *specfile* argument consisting of just a single line

dir `bin,bin 666 ----- ----- 0000...`

Integrity surveys the directory tree dependent from *rootdir*, or `/` if no *rootdir* is given. It reports non-bottom labels, which are possible signs of loss of integrity – modification without privilege. The search cuts off at directories with non-bottom labels.

SEE ALSO

getflab(2), *ftw*(3), *lcheck*(8)

BUGS

Extraneous diagnostics may be produced if this command is applied to active file systems.

NAME

nosh – ‘no-suprise’ shell, a sub-standard command interpreter

SYNOPSIS

`/etc/nosh` [*file*]

`priv nosh -gunxlp` *file*

DESCRIPTION

Nosh executes commands read from its standard input or from the named *file*. It has few of the advanced features of *sh*(1), making it more trustable for use in security administration tasks. In the second usage, *nosh* is endowed with one or more of the licenses `gunxlp`; see *labtoa*(3).

Commands

A command is either *simple* or *builtin.* Each command consists of a sequence of *words* separated by white space, terminated by a new-line character or end of input. Backslash quoting and sharp commenting are honored. The first word specifies the name of the command to be executed. If the command name matches one of the builtins listed below it is executed in the shell process. If the command name matches no builtin command, it is taken to be the pathname of an executable file; the name must begin with `/` or `..` A new process is created and an attempt is made to execute the file via *exec*(2) with an empty environment.

Input-Output Redirection

The standard input is inherited by simple commands. Simple > output redirection to named files as in *sh*(1) works only for simple commands, and only for file descriptors 1 (default) and 2.

Builtin Commands

`cd` *dir* Change the current directory to *dir*.

`exit` *status*

Exit with given status, 0 by default.

`set +e`

`set -e`

Turn an ignore-error switch on (`+e`, default) or off (`-e`). *Nosh* normally ignores nonzero exit status from an executed command, but exits with that status if `-e` is set.

`set +x`

`set -x`

Refrain from echoing (`+x`, default) or echo (`-x`) each command as it is executed.

`lmask` *licenses command* [*arg* ...]

Run a simple command, allowing licenses indicated by a nonempty string from the set `gunxlp-` to be inherited from *nosh.* Normally no licenses are inherited.

Missing features

Features of *sh*(1) that *nosh* lacks include: background commands, pipelines, compound commands, most builtins, multicharacter quotation, command substitution, parameter substitution, variables, environments, file name generation, redirection of input, signal traps, search paths, mail notification, `.profile`, user specification of prompts.

DIAGNOSTICS

Nosh prints nonzero exit or termination status of executed commands as octal numbers labeled `e=` and `t=`; see *wait*(2). If invoked with a *file* argument, it exits unconditionally for nonzero termination status or syntax error, and conditionally (under control of `set`) for nonzero exit status.

Nosh exits immediately if invoked with more than one argument, if invoked with an argument with a relative path name, if invoked by a relative path name, or if invoked with interrupt or quit signals ignored.

SEE ALSO

sh(1)

NAME

privserv – privilege server

SYNOPSIS

`lmask nuxl /etc/privserv` [*option* ...]

DESCRIPTION

Privserv is the keeper and interpreter of the *privs*(5) file. *Priv (1)* calls on *privserv* to hand out privileges in accordance with the rules given in *privs*. *Privserv* is a permanent process, normally started by the boot script *rc*(8). It receives service requests through the mounted pipe `/cs/priv`. The options are

`-p` *name*
: The file name of the server, `/etc/privserv` by default (used to reinvoke the priv server when the *privs*(5) file is modified by a `PRIVEDIT` request.)

`-m` *mountpt*
: The file system mount point for privilege service, `/cs/priv` by default.

`-l` *logfile*
: The file in which to record logging information, `/usr/adm/privlog` by default.

`-f` *privs*
: The data base of privileges, `/etc/privs` by default. Unless *privs* is itself a privileged file, *privserv* will not actually grant the privileges there specified.

FILES

`/etc/privs`
`/cs/priv`

SEE ALSO

priv(1)

NAME

pwserv – password verification service

SYNOPSIS

`/etc/pwserv`

DESCRIPTION

Pwserv, normally started from *rc*(8), handles password verification requests initiated by (say) *pwquery*(3) through the conventional process mount point `/cs/pw`. When a request is made a file descriptor (called the 'line' below) is passed to *pwserv* together with a user name and an optional parameter string. Normally, *pwserv* writes a prompt on the line, reads a reply, and returns an indication of success to the invoking client. Valid passwords are taken from the file `/etc/pwfile`, which lists for each user an ordinary (encrypted, *crypt*(3)-style) password and an SNK (Secure Net Key) challenge-response key. Before prompting, an `FIOPX` IO control is attempted to render the line to the end user private; see *pex*(4). If this succeeds either a classical or an Atalla password is accepted. If the pex bid fails, the prompt warns that the line is not private, and only an SNK response is accepted.

In the pexed case the prompt looks like `Password(pjw:31416):` and in the unpexed case like `Password(TAPPED LINE:01492):` The five digit string after the colon is the Atalla challenge string. Only the first five digits of the Atalla response string are significant. Hex digits in the response must be typed in lower case.

Possible values of the optional parameter string are

`pex` (specified by opening the server with `ipcopen("/cs/pw!pex") )` Accept passwords only if the `FIOPX` succeeds.

When the line's stream identifier asserts previous confirmation of the same password, *pwserv* answers affirmatively without demanding a password; see *session*(1) and *src*(5).

FILES

`/etc/pwserv`
`/etc/pwfile`

SEE ALSO

pwquery(3), *ipc*(3), *pex*(4), *stream*(4), *pwfile*(5), *passwd*(1)

BUGS

Jammable.

NAME

syslog, logpr – system security logging

SYNOPSIS

`priv syslog` *command* [*arg2* [*arg3*]]

`/etc/logpr` *file* [*offset*]

DESCRIPTION

Syslog controls the mandatory logging scheme. License `T_LOG` is required. The variety of different commands and command formats reflects the full complexity of the protean *syslog*(2) system call. In the usages given below a *mask* argument is a combination of letters `NILESDATUPX`, meaning:

`N` Record all uses of file names.
`S` Record all seek calls.
`U` Record all writes to the 'u area'.
`I` Record all accesses of inode contents.
`D` Record possession and use of file descriptors.
`P` Record process history: *exec*(2), *fork*(2), *kill*(2), *exit*(2).
`L` Record all explicit changes of labels by *setflab* (see *getflab*(2)) and *setplab* (see *getplab*(2)).
`A` Record all changes of labels.
`X` Record all uses of privilege.
`E` Record all `ELAB` error returns.
`T` Record all uses of a traced file or process.

Valid arguments to *syslog* are:

`on` *file logdev*
Nominate *file* as repository for user generated logging records written to logging special file *logdev*. *File* must be a full path name, and must be openable for writing. If *logdev*'s minor device number is zero, *file* will also receive mandatory (kernel generated) logging records. *Logdev* may be a full path name or a minor device number.

`off` *logdev*
Cancel the effect of an `on` command.

`get` *n* Print the value of the *n*-th log mask. Values of *n* are 0, 1, 2, or 3 for the 'poison' masks; 4 is 'global' mask.

`set` *n mask*
Set the value of the *n*-th log mask.

`fget` *file*
Print the poison level of *file*, one of the integers 0, 1, 2, or 3. *File* must be the full path name of a readable file.

`fset` *file n*
Set the poison level of *file* to *n*. *File* must be the full path name of a readable file.

`pget` *pid*
Print the logging mask of process *pid*.

`pset` *pid mask*
Set the logging mask of process *pid* to *mask*.

Logpr converts to cryptic ASCII the cryptic binary format of a log file described in *log*(5). The optional numerical byte offset tells where in the file printing is to start.

FILES

`/dev/log/log00` where *syslog* makes voluntary entries

SEE ALSO

syslog(2), *log*(4), *log*(5).

DIAGNOSTICS

'Covert channel warning': the log file has a label that is neither top nor flagged `L_NO`.

BUGS

Logpr is very primitive.

NAME

xs – checksums

SYNOPSIS

`xs` [-s] [-k *keystring*] [-f *official-list*] *file*...

DESCRIPTION

Xs computes and reports checksums of named files, one report per line, in the form

filename s1 s2 s3 s4

where the checksum comprises four groups of four hex digits each. The checksum algorithm may be perturbed by specifying a *keystring* argument. The `-s` argument causes the file's mode, label, owner and group to enter into the checksum calculation.

The `-f` argument causes *xs* to verify checksums of files against values given in the *official-list* file, which has the format of the output of an earlier *xs* run: lines consisting of one file name followed by four groups of hex digits per line. Text after a # sign is ignored.

The checksum algorithm used is meant to be secure: to create a file whose checksum agrees with that of another given file is very difficult.

EXAMPLES

```
xs -s ‘find /bin -print‘ | xs /dev/stdin
```

This should return a different value if `/bin` changes in any way.