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	
$\mathbf{y}$	label yes	§2.2
$\mathbf{n}$	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 ≤, the meet operation inf, the join operation sup, bottom element ⊥ and top element ` . The only comparison predicates we use are =, ≤, and ≰. The predicate ≰ means ‘‘not ≤’’ or ‘‘is not dominated by’’; it should not be thought of as >.

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

☞ Up describes a ≤ relation and down describes ≰. 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 ≤ to $\mathbf{L}^*$: for all x in $\mathbf{L}^*$, $x \le \mathbf{y}$, $\mathbf{y} \le x$, $\sup(x, \mathbf{y}) = \inf(x, \mathbf{y}) = x$, and $\sup(x, \mathbf{n}) = \inf(x, \mathbf{n}) = \mathbf{n}$. For all x in $\mathbf{L}$, $x \not\le \mathbf{n}$ and $\mathbf{n} \not\le x$. Also $\mathbf{n} \not\le \mathbf{n}$. Note that $(\mathbf{L}^*, \le)$ is not a lattice, and that ≤ is only partially transitive on $\mathbf{L}^*$ in that $L_1 \le L_2$ and $L_2 \le L_3$ imply $L_1 \le L_3$ 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 $\mathbf{L}$*-valued label, $L(x, t)$, written $L(x)$ when t doesn't matter. Labels of processes and seek pointers are restricted to $\mathbf{L}$.

Each participant x in data flow has a time-varying $\mathbf{L}$*-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) = $ ` .

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 $t_0 \le t_1$. 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 $x \to y$ occurs at time t, it must be upward.

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

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

Monotone labels. A persistent flow $x \to x$ must be upward unless $L(x, t_1) = \mathbf{n}$. so it is harmless to relabel a file **n**.

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

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

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

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

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.

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

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.

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

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

☞ 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.

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

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

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

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.

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

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 $t_0 \le t \le T_1$.

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

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

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)
Cap_{log}	$Cap_{log}(p)$ must be true
Cap_{extern}	$Cap_{extern}(p)$ must be true
Cap_{uarea}	$Cap_{uarea}(p)$ 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)		$f \rightarrow p$	READ(d)	i §3.5.13
		$[f \rightarrow s]$		
		$[s \rightarrow p]$		
dup(d)				u §3.5.6
dup2(d_1,d_2)				u §3.5.6
exec(f,a)		$p \rightarrow q$	RD(f)	iup §3.5.7
		$f \rightarrow q$		
		$u(p) \rightarrow u(q)$ *		
exec(f,0)		$f \rightarrow q$	RD(f)	iup §3.5.7
		$u(p) \rightarrow u(q)$ *		
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)	Cap_{extern}	$p \rightarrow f$	RD(f),W(f)	i §3.5.8
fmount5(n,d,f,n,c)	Cap_{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)		$p \rightarrow s$ $s \rightarrow p$	P(f),WS(d),RS(d)	§3.5.10
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)	$[p \rightarrow s]$	$f \rightarrow p$ $[f \rightarrow s]$ $[s \rightarrow p]$	P(f),READ(d)	§3.5.13
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)	Cap_{uarea}	$p \rightarrow u$		
setgroups(n,b)			X	
setlogname(b)	Cap_{uarea}	$p \rightarrow u$		
setpgrp(p,0)			u §3.5.9	
setpgrp(p,n)	Cap_{uarea}	$p \rightarrow u$		§3.5.9
setplab(l,c)				u §3.6.7
setruid(n)	Cap_{uarea}	$p \rightarrow u$		up
setuid(n)	Cap_{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)	$\mathrm{Cap}_{\mathrm{uarea}}$	$p \rightarrow u$		
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

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

$L(p) \in \mathbf{L}$

$C(p) \in \mathbf{L}$

$L(s) \in \mathbf{L}$, 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

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

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

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

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

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

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

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

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

Otherwise, if the critical inequalities hold the check succeeds.

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

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

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

Otherwise, establish the critical inequalities:

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

If $L(s) \ne M$ 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

$$
\begin{aligned}
&L(p) \le C(f)\\
&L(s) \le C(f)\\
&L(f) \le C(p)\\
&L(s) \le L(f)\\
&L(s) \le C(p)\\
&L(p) \le L(s)\\
&L(p) \le L(f)
\end{aligned}
$$

☞ 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 $L(f) = \mathbf{y}$.

☞ `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

$$
\begin{aligned}
&L(f) \le C(f)\\
&L(f) \le L(p)\\
&L(p) \le C(p)
\end{aligned}
$$

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 $F(p) \ne$ **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 $F(x) \ne$ **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 $L(p) \le L(f)$ 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 $L(f) \le L(q)$ 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 $L(f) \le C(q)$ 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. $L(\mathrm{High}) \not\le L(\mathrm{Low})$, 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 $APX(f) = \mathbf{false}$ by default, otherwise $APX(f) = \mathbf{true}$ 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 $\mathrm{Cap}_{\mathrm{log}}$. 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 $L(p) \le L(f)$, 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 $L(q) := \perp$.

If $L(p) \neq \perp$ 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 $\perp$. 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 $L(f) := \perp$, 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 $\perp$. 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 WS(*d*) (§3.1.8) as if $L(s)$ were equal to $\perp$.

If $n = 1$ check WS(*d*).

If $n = 2$ check 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 $L(f) \not\leq C(p)$ 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 $L(q) \not\leq L(p)$, 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 $\mathrm{Cap}_{\mathrm{nochk}}$ 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 $\mathrm{Cap}_{\mathrm{setpriv}}(p)$ 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 $L = \mathbf{y}$ then **error**.

Otherwise, if $L = \mathbf{n}$

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

Otherwise **error**.

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

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

Otherwise, if $\mathrm{Cap}_{\mathrm{nochk}}(p)$ 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 $F(f) = \mathbf{constant}$ then **error**.

Otherwise, if $F = \mathbf{constant}$ then **error**.

Otherwise, if $F = \mathbf{rigid}$ and *f* is not a stream then **error**.

Otherwise, if $F(f) = \mathbf{loose}$ the check succeeds.

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

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

Otherwise **error**.

If the fixity check succeeds, change the label:

If $F(f) = $ **rigid** then set $F :=$ **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 = $ **y** or if $L = $ **n** or if $C = $ **y** or if $C = $ **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 = $ **rigid** or $F = $ **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`	new state `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 $APX(f) := \mathbf{true}$.

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

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).