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.