NAME

getlab – print security labels of files and processes

SYNOPSIS

`getlab` [`-d`] [*file* ...]

DESCRIPTION

If there is a *file* argument, *getlab* prints, in the style of *labtoa*(3), the security labels of the named files. Otherwise, *getlab* prints the security label of the process and the process ceiling, The option is

`-d` Also print labels of all open file descriptors.

If a (character special) *file* can be opened, and the labels of the file and the file descriptor differ, both are printed.

EXAMPLES

`getlab /dev/stdin`

Print the labels (file system entry and file descriptor) of the standard input.

`drop getlab -d`

Print the process label, preventing the open-file check and the ceiling-label check (see *getplab*(2)) from raising the process label.

SEE ALSO

stat(1), *getflab*(2), *getplab*(2)

NAME

sign, enroll, verify, key, notaryd – sign and verify certificates

SYNOPSIS

`notary sign`

`notary enroll` [`-n`] *name*

`notary verify` *name xsum text*

`lmask xn /usr/notary/notaryd` [`-m` *mtpt*] [`-d` *dir*]

`notary key`

DESCRIPTION

Notary provides a document-authentication service. Any user may ‘sign’ a document by presenting it and a secret key to the notary. The notary returns a certificate (a cryptographic checksum made with the secret key). For the certificate to be useful, the key must be enrolled with the notary under some public name. Given the certificate and the public name, any user may ask the notary to authenticate the document by verifying that it is indeed as certified.

Sign writes on the standard output a certificate for its standard input. The secret key is demanded from the terminal.

Enroll prompts the terminal for a secret key to associate with the public *name*. Unless this is a new enrollment for *name*, indicated by option `-n`, the previous value of the key is demanded from the terminal. If a trivial new key is presented, the *name* is erased from the database.

Verify tells whether *xsum* is the checksum of *text*, figured with the enrolled key for the public *name*.

Notaryd is the notary daemon, which mounts itself on *mtpt* (default `/cs/notary`) and keeps its log files and database in directory *dir* (default `/usr/notary`). The database is encrypted, so that although *notaryd* is normally started by *rc*(8), it cannot serve other requests until it has been primed by a `notary key` request, which obtains the notary’s master key from the terminal.

FILES

`/cs/notary`
`/usr/notary/*`

SEE ALSO

notary(3)

NAME

passwd, pwx – change login password

SYNOPSIS

`passwd` [`-an`] [*name*]

`priv pwx` [[`-qcd`] *name*]]

DESCRIPTION

Passwd changes a password associated with the user *name* (your own name by default).

The program prompts for the old password and then for the new one. The caller must supply both. The new password must be typed twice, to forestall mistakes.

New passwords must be at least four characters long if they use a sufficiently rich alphabet and at least six characters long if monocase. These rules are relaxed if you are insistent enough.

Only the owner of the name or the super-user may change a password; the owner must prove he knows the old password.

If the `-a` option is given, *passwd* prompts for new values of certain fields of the password file entry.

The super-user may use the `-n` option to install new users. The prompts are self-explanatory, and most of the defaults obvious. A null response to the `UID:` prompt assigns a numeric userid one greater than the largest one previously in `/etc/passwd`. A null response to `Directory:` assigns a home directory in `/usr`. If the first character of the response to this prompt is an asterisk, the remaining characters are taken as the name of the new user's home directory, and a symbolic link to this directory is placed in `/usr`.

A new user's home directory starts with a file named `.profile`, which is a copy of `/etc/stdprofile` with `\N` replaced by the user's name, and `\D` replaced by the name of the user's home directory.

Pwx modifies the password entry for the named user in the secret password file, *pwfile*(5). With no option *pwx* changes the classical password for the named user, or the invoker by default. The options are

`-c` Change other information. A special editing password for a fictitious user, 'pwedit', is demanded. Then *pwx* prompts for treatment of the user password, SNK key, maximum privilege, and clearance (maximum ceiling).

`-d` Delete an entry. The editing password is demanded.

`-q` Demand the user password. If a correct password is entered, return status 0; otherwise nonzero.

Options `-c` and `-d` require `T_SETPRIV` privilege.

FILES

`/etc/passwd`
`/etc/stdprofile`
`/etc/pwfile`

SEE ALSO

crypt(3), *passwd*(5), *pwfile*(5)
Robert Morris and Ken Thompson, 'UNIX password security,' *AT&T Bell Laboratories Technical Journal* 63 (1984) 1649-1672

BUGS

The password file information should be kept in a different data structure allowing indexed access.

NAME

pcopy – paranoid file copy

SYNOPSIS

[`priv`] `pcopy` [*input output*]

DESCRIPTION

Pcopy copies an input file to an output file preserving, if possible, file ownership, dates, and label. The copying is performed in such a way as to assure faithfulness even in the presence of interfering processes.

Privilege, obtained via *priv*(1), is required to reproduce privileged files. The user must be able to write the output file, and be able to read and write files with the label of the input file.

SEE ALSO

cp(1), *pex*(4)

NAME

priv, privedit – run a command with privileges

SYNOPSIS

`priv` [*option* ...] [*command arg* ...]

`priv privedit` *node changes*

DESCRIPTION

If a *command* is given, *priv* determines from the *privs*(5) file the most specifically matching `REQUEST` for which the process has all the `NEEDS` and to which it has `ACCESS` (terminology explained in *privs*(5)). If a unique most specific match is found, *priv* asks for confirmation. Then, if the confirmation is `y`, the request is executed. Privileges and process ceiling are set according to the pertinent entry in `/etc/privs` and the current directory is set to a place with security label `L_NO`; see *getflab*(2). Thus relative pathnames won't work in the *command* until it executes *chdir*(2).

If no command is given, the contents of the *privs* file are printed on the standard output.

The options are

`-n` Determine and report authorization and actions. Do not execute them except, if `PRIVEDIT` is requested, place the edited privilege file on the standard output.

`-f` *servfile*
Use *servfile* instead of `/cs/priv`, to use a non-standard privilege server.

One request is more specific than another if the regular language for each argument of the first request is contained in the corresponding language for the second request, and at least one containment is proper.

The standard error and standard input are used for confirmations. Both must come from the same trusted source, either a pexable stream with a stream identifier, or a pipe from a trusted process; see *pex*(4) and *stream*(4).

Privedit applies to the *privs* file the modifications given in the *changes* file. Only the part of the authorization tree rooted at the given *node* may be changed. The form of *changes* is described in *privs*(5). The changes are echoed and confirmation is requested. (*Privedit,* like any other *command,* is a conventional token defined by the *privs* file; it is not built in.)

Priv clears the environment to prevent hidden corruption by untrusted processes. For the same reason it asks confirmation of the argument list. What you see is what it will do.

The real work of *priv* is done by *privserv*(8). *Priv* communicates with *privserv* via a pipe that the latter mounts on `/cs/priv`.

FILES

`/etc/privs`
`/cs/priv`

SEE ALSO

privs(5), *privserv*(8), *session*(1)

DIAGNOSTICS

If a *command* is performed, *priv* returns the result of the last constituent action; see *privs*(5).

BUGS

Trailing null *args* are deleted.
The standard input and standard error cannot freely be redirected.
It is possible for a password to be demanded twice. This would be mitigated if requests were assessed in decreasing order of specificity instead of table order.

NAME

redmail, blackmail – multilevel mail

SYNOPSIS

`redmail`

`blackmail` *maildir*

DESCRIPTION

These commands arrange for the delivery and receipt of mail with varying security labels.

Redmail simulates *mail*(1) for reading multilevel mail.

To receive timely notification of multilevel mail, set the shell variable `MAIL` to *maildir*`/FLAG`.

Blackmail delivers multilevel mail to the private mail directory *maildir*. To arrange for incoming mail to be handled by *blackmail*, provide a directory *maildir* (conventionally `.mail` in your home directory) and edit a single line like

```
Pipe to blackmail $HOME/.mail
```

into your regular mailbox, `/usr/spool/mail/`*logname*. Thereafter a separate *blackmail* process runs for each letter. Letters receive the security label of their source.

FILES

`$HOME/.mail/*`

SEE ALSO

mail(1)

NAME

session, drop, runlow – substitute labels temporarily

SYNOPSIS

`session` [*option* ...]

`priv session` [*option* ...]

`runlow` *command*

`drop` [`-l` *label*] [*command-arg* ...]

DESCRIPTION

Session sets a temporary security label for the duration of one command. The ceiling is raised sufficiently to cover the requested label, up to the authorization recorded for the current login name. If no *command-args* are given, the command is taken to be a shell: *sh*(1) above the system floor, or *nosh*(8) below. With *command-args,* the specified command is run; there is no shell-like path search.

If the current ceiling does not dominate the new ceiling, or the the new process label is below the system floor and does not dominate the current label *session* must be invoked through *priv*(1).

The options are

`-l` *label*
: Set the process label and the label of the standard input to the given value, specified as in *atolab*; see *labtoa*(3). If the value does not dominate the current process label, clear the environment and pass no arguments to the invoked command. If *label* is missing, it is taken to be the system floor.

`-C` *label*
: Set the process ceiling at or above the given value. If *label* is missing, it is taken to be the process label.

`-u` *user*
: The password for *user* will be demanded. The fact that the password has been presented will be recorded in the stream identifier (see *stream*(4)) of the standard input. For the duration of the session, further queries for that password will succeed automatically. If *user* is missing, it is taken to be the current login name.

`-x`
: Replace current session instead of suspending it for the duration of the new session (like `exec` in *sh*(1)).

`-c` *command-arg ...*
: Instead of a shell, run the given command with the given arguments. This option must come last.

To change labels, the input source must come over a trustable channel, in particular neither from an untrusted computer nor from a terminal into which untrusted code has been downloaded. The request may require confirmation to assure that no software has tampered with it; answer `y` for yes. Confirmation and password inquiries happen under cover of *pex*(4). In a *mux*(9.1) window, this gives a visible indication; a missing indication is a sign of spoofing.

Runlow runs a command, starting the label at bottom, somewhat like `session -l 0`, but without changing the label of the standard input. The executable file is located according to environment variable `$PATH` as in *sh*(1). The command receives empty argument and environment lists, but inherits open file descriptors; only descriptors 0-3 are allowed. The process label will immediately rise to dominate that of the executable file.

Drop sets the process ceiling to *label* (by default to the process label) for the running of one *command* with the given *arguments.* If no *command* is given, `/bin/sh` is run.

The current process label, process licenses, terminal label, and environment are preserved.

EXAMPLES

`priv session -C ffff...`
: Change ceiling to the maximum authorized for the current user.

```
priv session -l 0
cd /usr/src
```

Enter a bottom-label interactive terminal subsession. Get out of the black-hole directory that *priv*(1) leaves you in.

```
runlow /bin/sh    # not useful
```

An attempt to fool the system into giving a bottom-label interactive shell. When the shell reads from standard input, its label will revert to that of the current session.

```
drop ls -l *
drop pwd
```

Prevent the process label from rising to cover the labels of files in the directories examined by *ls* or *pwd.* (If the label did rise, the output could not get to the terminal.)

FILES

```
/dev/log/sessionlog
/etc/pwfile
/etc/floor
/bin/sh
/etc/nosh
```

SEE ALSO

sh(1), *getflab*(2), *getplab*(2), *exec*(2), *pwfile*(5), *login*(8), *nosh*(8), *pwserv*(8)

DIAGNOSTICS

'Sorry', instead of asking for a password: untrusted channel.

NAME

setlab, downgrade, setpriv – set security label on files

SYNOPSIS

`setlab` [*option* ...] *label file* ...]

`priv downgrade` [`-v`] *delta file* ...

`priv setpriv` *cap lic file* ...

DESCRIPTION

Setlab sets the security label on the named *files*, or on the standard input if no files are named. The *label* is a single argument in the style accepted by *atolab*; see *labtoa*(3). The options are

`-a` Add *label* to the current file label (*new*=*old* | *label*).

`-s` Subtract *label* from the current file label (*new*=*old*&˜*label*).

`-p` Set privileges (capabilities and licenses) only.

`-v` Print a blow-by-blow account on standard error file.

The process must be able to open the file, either for reading or writing. One or more licenses (see *getplab*(2)) are needed in some instances:

`T_EXTERN` to downgrade (new label does not dominate old)

`T_SETPRIV` if either the old or the new label has nonzero privilege bits

`T_NOCHK` if the old label has flag `L_NO` (also need `T_EXTERN` to change away from `L_NO`).

Downgrade uses *setlab* to clear the label bits designated by *delta*. It is a conventional request defined in the privilege file, *privs*(5), which checks that the user has authority over the specified label bits and supplies the necessary privilege to *setlab.*

Setpriv is a conventional interface to *setlab* for changing file capabilities and licenses.

EXAMPLES

`setlab ffff... file`
Give the file a top label.

`setlab -a F file`
Freeze a file label to keep writes from raising the lattice value.

`lmask x setlab -s 03 file`
Downgrade a security label using a privileged *nosh*(8) session. The *downgrade* priv request is preferred.

`priv downgrade 03 file`
Same, using obtaining the necessary authorization and privilege from *priv*(1).

`priv setpriv - n file`
Give the file a license, but no capabilities. This is a conventional trick to make the file immutable until its privileges are turned off again. The lattice value of the label is bottom (all zero).

DIAGNOSTICS

‘Locking file for vetting’. As a matter of policy, *setlab* refuses to assign arbitrary privileges to a previously unprivileged (‘untrusted’) file. Instead it marks the file immutable as in the last example. The file may then be examined at leisure to assess whether its contents are indeed trustable before privileges are finally assigned.

SEE ALSO

getflab(2), *getlab*(1), *priv*(1)

BUGS

The strings `-a` and `-p` happen to be legitimate, if unusual, labels. They will always be understood as option flags.

NAME

stat – file statistics and labels

SYNOPSIS

`stat` *file* ...

DESCRIPTION

Stat places facts about the named *files* on the standard output. Successive output lines show

The file name.

Inode number, mode, link count, owner, group, and size displayed like output from *ls*(1) with options `lidL`. For device files, the size is replaced by major and minor device numbers separated by a comma.

The major and minor device numbers of this inode's file system and the file mode in octal.

Modification, access, and change times, each on a separate line.

The security label (of the destination, if a symbolic link) is given in the style of *labtoa*(3).

If the file can be opened and corresponding data differ for the opened file, similar information for the opened file follows.

If the file is a symbolic link, the link destination is given, marked by `->`.

Stat has nocheck capability; a superuser with nocheck license can use it to examine any file.

EXAMPLES

```
/dev/tty:
      0 crwxrwxrwx 0 root 0 0,0
      255,255 020777
      Jun 22 22:52:30 1988
      Jun 22 22:52:30 1988
      Jun 22 22:52:30 1988
      ------ ------CY 0000 0000 ...
      974  rw-rw-r-- 0 reeds other 0
      255,255 0100664
      ------ ------R  0000 0000 ...
/usr/spool/man:
      9926 lrwxrwxrwx 1 doug bin 23
      7,66 0120777
      Oct 17 21:21:21 1987
      Jun 22 22:52:14 1988
      Oct 17 21:21:21 1987
      ------ ------C  0000 0000 ...
      -> /n/bowell/usr/spool/man
```

DIAGNOSTICS

Diagnostics appear on the standard output.

SEE ALSO

stat(2), *ls*(1), *getlab*(1)

NAME

intro, fmount, getuid, signal, stat, wait – changes to manual

SYNOPSIS

```
#include <sys/label.h>

int fmount5(type, fildes, name, flag, ceiling)
char *name;
struct label *ceiling;
```

DESCRIPTION

This section covers small changes in the named manual pages for IX relative to v10.

intro

New error returns:

38 `ELAB` Security label violation
: An action which would, if completed, break security rules; see *getplab*(2).

39 `ENOSYS` No such system call
: An attempt to execute a nonexistent or unsupported system call.

40 `ENLAB` Out of security labels
: A system table for security labels is full: a trouble similar to `ENFILE`.

41 `EPRIV` Insufficient privilege
: An attempt was made to execute a priviledged system call, or exercise a privileged feature of a regular system call.

fmount

Fmount5 mounts a file system as does *fmount*(2) and, on regular (type 0) or network (type 4) file systems, imposes a specified *ceiling* label. No file in the file system can be accessed unless the label of the file is dominated by the file system ceiling. Moreover, in determining capabilities during *exec*(2), capability and license bits in the file label are masked by corresponding bits in the file system ceiling. The default ceiling is `L_YES` on regular and `L_NO` on network file systems. Default capabilities and licenses are all zero.

getuid

Whenever one of the system calls *setuid*, *setgid*, *setruid*, or *setlogname* requires superuser status, it also requires capability `T_UAREA`.

Setpgrp can set the process group only to the current process id unless the process has capability `T_UAREA`.

signal

Security label violations by *write*(2) result in `SIGPIPE`. Other security label violations result in `SIGLAB`, which is ignored if not caught.

stat

New modes. These are indicated in *ls*(1) by `a` and `b` appended to the usual mode field.

`S_IAPPEND`
: Append-only file.

`S_IBLIND`
: Blind directory. A blind directory cannot be read and is immune to security label checks on search; files can be removed from it only by their owners.

wait

If the security label of the waiting process does not dominate that of the exiting process, then nonzero termination status or exit code is reported simply as `SIGTERM`.

NAME

execl, execv, execle, execve, execlp, execvp, exect, environ – execute a file

SYNOPSIS

```
int execl(name, arg0, arg1, ..., argn, (char *)0)
char *name, *arg0, *arg1, ..., *argn;

int execv(name, argv)
char *name, *argv[];

int execle(name, arg0, arg1, ..., argn, (char *)0, envp)
char *name, *arg0, *arg1, ..., *argn, *envp[];

int execve(name, argv, envp)
char *name, *argv[], *envp[];

int execlp(name, arg0, arg1, ..., argn, (char *)0)
char *name, *arg0, *arg1, ..., *argn;

int execvp(name, argv)
char *name, *argv[];

int exect(name, argv, envp)
char *name, *argv[], *envp[];

extern char **environ;
```

DESCRIPTION

Exec in all its forms overlays the calling process with the named file, then transfers to the entry point of the image of the file. There can be no return from a successful *exec*; the calling image is lost.

Files remain open across *exec* unless explicit arrangement has been made; see *ioctl*(2). Signals that are caught (see *signal*(2)) are reset to their default values. Other signals' behavior is unchanged.

Each user has a *real* userid and groupid and an *effective* userid and groupid. The real userid (groupid) identifies the person using the system; the effective userid (groupid) determines access privileges. *Exec* changes the effective userid and groupid to the owner of the executed file if the file has the set-userid or set-groupid modes. The real userid is not affected.

The security label (see *getflab*(2)) of the process is set as follows. If any arguments or environment parameters are present, or if and file descriptor numbers greater than 3 are in use, the lattice value of the process label is ascribed to them, otherwise lattice bottom. This value is ORed with the lattice value of the executed file to obtain the new lattice value for the process. If the new lattice value does not dominate the old, the permission mask (see *umask*(2)) is set to 022.

Process licenses persist. In the simplest case, the process obtains from the file the capabilities for which the process has licenses; see *getplab*(2). The detailed computation for process capabilities is: Nominal capabilities are determined by ANDing the file capabilities with the capabilities in the file system ceiling (see *mount*(2)) and then ORing with built-in minima. Nominal licenses are determined by ANDing the file licenses with the licenses in the file system ceiling and with built-in maxima. Process capabilities are set by ORing the process licenses with the nominal licenses, then ANDing with the nominal capabilities.

The builtin minimum file capabilities are all 0. The builtin maximum file licenses for `T_SETPRIV` and `T_LOG` are 0; the rest are 1.

The *name* argument is a pointer to the name of the file to be executed. If the first two bytes of that file are the ASCII characters `#!`, then the first line of the file is taken to be ASCII and determines the name of the program to execute. The first nonblank string following `#!` in that line is substituted for *name.* Any second string, separated from the first by blanks or tabs, is inserted between the first two arguments (arguments 0 and 1) passed to the invoked file.

The argument pointers *arg0, arg1, ...* or the pointers in *argv* address null-terminated strings. Conventionally argument 0 is the name of the file.

Execl is useful when a known file with known arguments is being called; the arguments to *execl* are the character strings constituting the file and the arguments.

Execv is useful when the number of arguments is unknown in advance; the arguments to *execv* are the name of the file to be executed and a vector of strings containing the arguments. The last argument string must be followed by a 0 pointer.

When a C program is executed, it is called as follows:

```
main(argc, argv, envp)
int argc;
char **argv, **envp;
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. Conventionally *argc* is at least 1 and `argv[0]` points to the name of the file.

Argv is directly usable in another *execv* because `argv[argc]==0`.

Envp is a pointer to an array of strings that constitute the *environment* of the process. Each string conventionally consists of a name, an `=`, and a null-terminated value; or a name, a pair of parentheses `(),` a value bracketed by `{` and `}`, and a null character. The array of pointers is terminated by a null pointer. The shell *sh*(1) passes an environment entry for each global shell variable defined when the program is called. See *environ*(5) for some conventionally used names.

The C run-time start-off routine places a copy of *envp* in the global cell *environ*, which is used by *execv* and *execl* to pass the environment to any subprograms executed by the current program. The *exec* routines use lower-level routines as follows to pass an environment explicitly:

```
execve(file, argv, environ);
execle(file, arg0, arg1, . . . , argn, (char *)0, environ);
```

Execlp and *execvp* are called with the same arguments as *execl* and *execv*, but duplicate the shell's actions in searching for an executable file in a list of directories given in the `PATH` environment variable.

Exect is the same as *execve,* except it arranges for a stop to occur on the first instruction of the new core image for the benefit of tracers, see *proc*(4).

FILES

`/bin/sh` shell, invoked if command file found by *execlp* or *execvp*

SEE ALSO

fork(2), *environ*(5)

DIAGNOSTICS

`E2BIG`, `EACCES`, `EFAULT`, `EIO`, `ELAB`, `ELOOP`, `ENOENT`, `ENOEXEC`, `ENOMEM`, `ENOTDIR`, `ENXIO`, `EROFS`, `ETXTBSY`

BUGS

If *execvp* is called to execute a file that turns out to be a shell command file, and if it is impossible to execute the shell, some of the values in *argv* may be modified before return.
The path search of *execlp* and *execvp* does not extend to names substituted by `#!`.

NAME

getflab, fgetflab, setflab, fsetflab – get or set file security label and privilege

SYNOPSIS

```
#include <sys/label.h>

getflab(name, labp)
char *name;
struct label *labp;

fgetflab(fildes, labp)
struct label *labp;

setflab(name, labp)
char *name;
struct label *labp;

fsetflab(fildes, labp)
struct label *labp;
```

DESCRIPTION

Getflab copies the security label from the the named file into the structure pointed to by *labp*. *Fgetlab* copies the security label from an open file specified by file descriptor. The field `lb_junk` is always zero.

The structure of a security label as defined in `<sys/label.h>` is

```
#define LABSIZ          60
struct  labpriv {
        unsigned int    lp_junk : 16,/* poison level, see syslog(2) */
                        lp_flag : 2,
                        lp_fix : 2,/* fixity */
                        lp_t : 6,/* capabilities */
                        lp_u : 6;/* licenses */
};
struct  label {
        struct labpriv lb_priv;
        unsigned char  lb_bits[LABSIZ];
#       define lb_junk lb_priv.lp_junk
#       define lb_flag lb_priv.lp_flag
#       define lb_t    lb_priv.lp_t
#       define lb_u    lb_priv.lp_u
#       define lb_fix  lb_priv.lp_fix
};
                        /* codes in lb_flag */
#define L_YES           1
#define L_NO            2
#define L_BITS          3
                        /* codes in lb_fix */
#define F_LOOSE         0
#define F_FROZEN        1
#define F_RIGID         2
#define F_CONST         3
                        /* bits of lb_t and lb_u */
#define T_SETPRIV       001     /* may set file privilege */
#define T_SETLIC        002     /* may change process license */
#define T_NOCHK         004     /* exempt from label checking */
#define T_EXTERN        010     /* may introduce foreign data */
#define T_UAREA         020     /* may write in u area */
```

```
#define T_LOG          040     /* may execure syslog() call */
```

Three types of labels are distinguished by the `lb_flag` field:

`L_YES` The file can be read or modified without regard to label. Its inode data (see *stat*(2)) have permanent conventional values. *Null*(4), *log*(4), and *fd*(4) are labeled `L_YES`.

`L_NO` The the file and its inode cannot be read or written except by processes with capability `T_NOCHK`. A `L_NO` label may be changed by processes with capability `T_EXTERN`, unless prevented by `F_CONST` described below.

`L_BITS` The label has a 'lattice value', given by `lb_bits` and so called because the values form a mathematical lattice with bitwise AND as the meet operation and OR as the join.

Each process and each file has a label. Normally data may only flow 'up' the lattice. The destination of a read, write, inode query, or inode change must have a lattice value that dominates (bitwise) the lattice value of the source, unless the process concerned has capability `T_NOCHK`.

To assure upward flow, a *read*(2) or an inode query (e.g. *stat*(2)) normally causes the file label to be OR-ed into the process label. Similarly a *write*(2) or an inode change (as by *chmod*(2) or *link*(2)) causes the process label to be OR-ed into the file label. However such side-effect changes in a file or process label may happen only if the label is loose (see below) and the new label is dominated by the process ceiling; see *getplab*(2). Otherwise the system call terminates with error `ELAB`.

Security checks are independent of, and made prior to, the permission checks described in *access*(2). Super-user processes are subject to security checks.

Setflab replaces the security label of the named file with the contents of the structure pointed to by *labp*. *Fsetflab* replaces the security label of an open file specified by file descriptor. If the new label has flag `L_BITS`, the new lattice value must dominate the old one, dominate the process label, and be dominated by the process ceiling. If the new label has flag `L_NO`, the old label must be dominated by the process ceiling. Flag `L_YES` is an error. The field `lb_junk` is ignored.

The field `lb_t` contains 'capability' bits; `lb_u` contains corresponding 'license' bits; their meanings are described in *getplab*(2) and *exec*(2). The two fields together are known as 'privileges'. Any file that has nonzero privileges is called 'trusted' and cannot be changed, in contents or in inode, except by processes with capability `T_SETPRIV`.

Aside from considerations of trustedness, a label can be changed with more or less freedom according to its 'fixity', `lb_fix`:

`F_LOOSE` Any process can change the lattice value of a loose file label implicitly as a side effect as described above or (up to the process ceiling) explicitly with *setflab* or *fsetflab*. The file owner or the super-user can change the fixity.

`F_FROZEN` The lattice value of a frozen label cannot change. The fixity can be changed by the file owner or the super-user.

`F_RIGID` Only processes with capability `T_EXTERN` can change a rigid label; see *getplab*(2). The labels of external media, such as terminals, tapes or disks, are automatically rigid. A loose or frozen label on a stream (see *stream*(4)) can be changed to rigid. This facility allows filters, such as *mux*(9.1), to make pipes behave like external devices. The fixity of a rigid label cannot change.

`F_CONST` A constant label may not be changed. The labels of certain special files, such as `/dev/null` and `/dev/mem`, are automatically constant; no other labels may become constant.

SEE ALSO

getplab(2), *getlab*(1), *labLE*(3), *setlab*(8), *unsafe*(2), *signal*(2)

DIAGNOSTICS

EFAULT, EIO, ELAB, ELOOP, ENOENT, ENOTDIR

NAME

getplab, setplab – get or set process security label and privilege

SYNOPSIS

```
#include <sys/label.h>

getplab(labp, ceilp)
struct label *labp, *ceilp;

setplab(labp, ceilp)
struct label *labp, *ceilp;
```

DESCRIPTION

Getplab copies the security label and the ceiling label, usually simply called 'the ceiling', of the current process into the structures pointed to by *labp* and *ceilp.* No copy happens for a zero pointer. The structure and meaning of labels are described in *getflab*(2). The ceiling is a security lid; the process can only access files with labels dominated by the ceiling.

A process may have special security 'capabilities', in which case it is called 'trusted'. The capabilities are obtained from the file it is executing, usually as 'licensed' from its parent process; see *exec*(2). The capabilities and corresponding licenses are given by bits in the fields *labp*`->lb_t` and *labp*`->lb_u` respectively. The bits are defined by the masks

`T_SETPRIV` The process can change the privileges of files; see *getflab*(2).

`T_SETLIC` The process can increase its own licenses; see below.

`T_EXTERN` The process can bring new data sources into view by mounting file systems or setting labels of (open) special files; see *getflab*(2).

`T_NOCHK` Ordinary checks and changes of lattice values are not made when reading or writing files or inodes or when setting the process label.

`T_UAREA` The process can change certain information that may be accessed by descendent processes without label checks; see *setuid*(2) and *stream*(4).

`T_LOG` The process can change logging status; see *syslog*(2).

Setplab copies the structures pointed to by *labp* and *ceilp* into the process label and the ceiling label. Unless the process has capability `T_NOCHK`, the new lattice value of the process label must dominate the old and the old lattice value of the ceiling must dominate the new.

The new label flag must be `L_BITS`, and the lattice value of the new ceiling label must dominate the lattice value of the new process label.

Capabilites may not increase. Licenses may increase only if the process has capability `T_SETLIC`.

The fixity, `lb_fix`, of a process may be set only to `F_LOOSE` or `F_FROZEN`. In the latter case the process label can not change as a side effect of label checking.

The bits of the ceiling pointer are themselves labeled as if they were a minifile. When the ceiling is set by *setplab*, the minifile label is set to the old value of the process label, unless the process has capability `T_SETLIC`, in which the minifile label is set to bottom. When the ceiling is read by *getplab*, the minifile label is checked as if read by *read*(2).

DIAGNOSTICS

`EFAULT`, `ELAB`, `EPRIV`

If *getplab* cannot raise the process label to dominate the minifile label, the requested labels are filled in, with the ceiling being censored to flag `L_NO` , and `ELAB` is returned.

SEE ALSO

getflab(2), *unsafe*(2), *exec*(2), *session*(1), *setlab*(8)

NAME

labmount – return file system ceiling label

SYNOPSIS

```
int labmount(fd, lp)
struct label *lp;
```

DESCRIPTION

If the file with descriptor *fd* resides in a file system, *labmount* copies the ceiling label of that file system into the place pointed to by *lp*. If the file does not reside in a file system, the ceiling is reported to be `L_YES`; see *getflab*(2).

SEE ALSO

fmount(2)

DIAGNOSTICS

`EBADF, EFAULT, EIO, ELOOP, ENOENT, ENOTDIR`

NAME

nochk – control security checking by file

SYNOPSIS

```
nochk(fd, onoff);
```

DESCRIPTION

Nochk modifies file security checks in processes that have capability `T_NOCHK`. If *onoff* is 1, file descriptor *fd* becomes exempt from security checks; this is the default state. If *onoff* is 0, the file descriptor will be checked as if the process did not have capability `T_NOCHK`.

The return value is the previous checking state.

SEE ALSO

getplab(2)

DIAGNOSTICS

`EBADF`

BUGS

It would have been wise to let 0 be the default state, but this would have required modifying standard utilities, such as *fsck*(8), which must be run with privilege `T_NOCHK`.

NAME

seek, tell, lseek, llseek – manipulate read/write pointer

SYNOPSIS

```
int seek(fildes, offset, whence)
long offset

long tell(fildes)

long lseek(fildes, offset, whence) long offset;

Long llseek(fildes, offset, whence)
Long offset;
```

DESCRIPTION

Seek sets the file pointer for the file associated with *fildes* as follows:

> If *whence* is 0, the pointer is set to *offset* bytes.
>
> If *whence* is 1, the pointer is set to its current location plus *offset*.
>
> If *whence* is 2, the pointer is set to the size of the file plus *offset*.

Tell returns the value of the file pointer associated with *fildes.*

Lseek is equivalent to *seek* followed by *tell.*

Llseek is like *lseek*, but handles i.e. 64-bit, file pointers.

Seeking far beyond the end of a file, then writing, creates a gap or 'hole,' which occupies no physical space and reads as zeros.

File pointers have security labels separate from files. For security-label calculations, *seek* is understood to 'write' the pointer, *tell* to 'read' it. If *whence* is 0 on *seek*, the new value of the file pointer does not depend on the old value.

SEE ALSO

open(2), *fseek*(3)

DIAGNOSTICS

`EBADF`, `ESPIPE`

BUGS

Lseek doesn't affect some special files.

NAME

syslog – security logging

SYNOPSIS

```
#include <sys/log.h>

int syslog(command, arg2, arg3)
```

DESCRIPTION

Syslog controls security logging. The *command* argument determines the meaning of the other arguments.

Logging is done by writing on special files, described in *log*(4). One of these files is the 'system log file' where the kernel records certain events automatically. Each process has an 'audit mask' that determines which events cause logging records; mask items are defined in `<sys/log.h>`; see *log*(5). Each file has a 'poison class' with value 0, 1, 2, or 3. The kernel has a table of four corresponding 'poison masks' and a global audit mask. When a system call mentions a file in a pathname, the poison mask corresponding to the file's poison level is ORed into the process audit mask; when a process executes a file, the global log mask is ORed into the process audit mask.

The forms of the several *syslog* commands follow. Arguments shown as `0` are ignored.

`syslog(LOGON,` *fd* `,` *x* `)`
: Turn logging on; nominate file descriptor *fd* as repository for log file with minor device number *x*. *Fd* must be open for writing. Logging will persist after *fd* is closed.

`syslog(LOGOFF, 0,` *x* `)`
: Turn logging off on minor device number *x*.

`syslog(LOGGET,` *n*`, 0)`
: Return the value of the *n*th poison mask; *n*=4 designates the global audit mask.

`syslog(LOGSET,` *n*`,` *x*`)`
: Set the *n*th poison mask to *x*.

`syslog(LOGFGET,` *fd*`, 0)`
: Return the poison level of the file associated with file descriptor *fd*, which may be open for reading or writing.

`syslog(LOGFSET,` *fd*`,` *x*`)`
: Set the poison level of the file associated with file descriptor *fd*, (which may be open for reading or writing) to *x*. The poison level is stored in field `di_label.lb_junk` of the file's inode; see *inode*(5).

`syslog(LOGPGET,` *pid*`, 0)`
: Return the audit mask of process *pid*.

`syslog(LOGPSET,` *pid*`,` *x*`)`
: Set the audit mask of process *pid* to *x*.

Syslog works only in processes with capability `T_LOG`; see *getplab*(2).

SEE ALSO

log(4), *log*(5), *syslog*(8)

DIAGNOSTICS

`EBADF, EFAULT, EINVAL, EIO`

NAME

unsafe – detect potential file security violations

SYNOPSIS

```
#include <sys/types.h>

unsafe(nfd, readfds, writefds)
fd_set *readfds, *writefds;
```

DESCRIPTION

Unsafe examines file descriptors 0 through *nfd–1* and sets the corresponding bits of the masks in *readfds* and *writefds* to indicate files that are not known to be safe (i.e. to satisfy security label rules) for reading or writing respectively. The bit masks are indexed and manipulated as described in *select*(2).

At the same time, if the process has capability `T_NOCHK` (see *getplab*(2)), all file descriptors indicated by ones among the first *nfd* bits of the previous values of *readfds* and *writefds* are marked safe to read or write respectively.

Potentially unsafe situations arise from changes in file label caused by this or other processes, changes in process label, and file opening.

To prevent unintended violations of security policy, programs with capability `NOCHK` must monitor label changes. For this purpose the process label may be frozen (see *getplab*(2)) to prevent unintended automatic label changes. `SIGLAB` may be used to detect changes in file labels (see *signal*(2)), and *unsafe* to pinpoint them.

DIAGNOSTICS

`EFAULT`

SEE ALSO

getflab(2), *getplab*(2), *signal*(2), *select*(2)

NAME

buildmap, transin, transout – translate labels between computers

SYNOPSIS

```
#include <cbit.h>

struct mapping *buildmap(int fd, char *file, char *me, char *pw, server);

transin(struct mapping *map, struct label *foreign,struct label *domestic);

transout(struct mapping *map, struct label *domestic, struct label *foreign);
```

DESCRIPTION

Buildmap and its partner (which may be another instance of *buildmap*) at the far end of the stream *fd* work out a mapping for translating labels. The label space at the near end is defined by *file* , which contains ASCII representations of *cbit*(3) structures. The identity (for authorization purposes) of the near end is *me*, using a secret password *pw*. Each end challenges the other, using its own password to compose a response, whose validity is checked by *verify ;* see *notary*(3); The ends need not know each other's passwords. *Server* is to simplify the protocol: one end should have server set to zero, the other end to one.

Transout and *transin* translate labels according to the formula determined by *buildmap*. Labels in transit are represented in the binary form of the source machine and are translated on receipt, so *transout*'s job is simpler. Both routines return 0 if translation is impossible or illegal, otherwise they perform the translation and return 1.

SEE ALSO

cbit(3), *notary*(3)

DIAGNOSTICS

These routines all return 0 on error.

NAME

cbit, cbitread, cbitlookup, cbitparse, cbitcert – security category manipulation

SYNOPSIS

```
#include <cbit.h>

struct cbit *cbitparse(char **fields, struct cbit *cb);

struct cbit *cbitread(char *file);

struct cbit *cbitlookup(char *name, struct cbit *cb);

char *cbitcert(struct cbit *p);
```

DESCRIPTION

These functions manipulate certificates entitling computers to handle compartmented security categories. Each security compartment is represented by a structure of form:

```
struct cbit {
        char *name;           official name of category
        int floor;            default value (only bottom bit used)
        char *owner;          public name of issuing authority
        char *nickname;       our version of category name
        int bitslot;          where we store it
        char *exerciser;      who we are
        char *certificate;    owner's signature
}
```

which describes the meaning of the *bitslot*-th bit in a computer's label space. By convention, the lines of the file `/etc/cbits` contain (in ASCII colon-separated form) the compartments currently held on the local computer.

Cbitparse fills in and returns a cbit in the obvious way from a vector of seven strings. If the second argument is zero *cbitparse* allots a new structure using *malloc*(3).

Cbitread reads and parses an ASCII file of cbits, returning an array of filled in structures. The last entry in the array is a dummy; it is signalled by having a zero value of *name*.

Cbitlookup, when fed a category name and an array of cbits (such as returned by *cbitread*), returns a pointer to the unique entry whose category name is *name*, or returns zero.

Cbitcert composes a certificate granting *exerciser* the right to hold files with the given security category. The output of *cbitcert* depends only on *name*, *floor*, *owner*, and *exerciser*. The output must be signed by *owner* with *xs* (see *notary*(3)) to produce the checksum value in *certificate*. Third parties may check validity of a cbit by calling

```
verify(p->exerciser, p->certificate, cbitcert(p), strlen(cbitcert(p)))
```

FILES

`/etc/cbits`

SEE ALSO

notary(3).

DIAGNOSTICS

These routines all return 0 on error.

NAME

getstsrc, setstsrc – read and write a stream identifier

SYNOPSIS

```
char *getstsrc(fd)
setstsrc(fd, name)
char *name;
```

DESCRIPTION

Setstsrc attaches a descriptive string to the indicated stream, and *getstsrc* returns a pointer to a static buffer containing the string. The string persists until final close of the stream. When a stream is first opened the string is trivial.

Setstsrc requires capability `T_EXTERN`; see *getplab*(2). The string conventionally names the off-machine source of the stream. Since only trusted processes may modify it, it may be relied on for security calculations.

Getstsrc returns 0 on error, *setstsrc* returns –1 on error.

SEE ALSO

stream(4)

DIAGNOSTICS

`EPERM`, `ENOTTY`

BUGS

The return value of *getstsrc* points to static data whose content is overwritten by each call.

NAME

labelyes, labelno, labeltop, labelbot – label constants

SYNOPSIS

```
extern struct label labelyes;

extern struct label labelno;

extern struct label labeltop;

extern struct label labelbot;
```

DESCRIPTION

These objects are initialized as follows, where the coded values are as in *labtoa*(3).

`labelyes`	The universally permissive label, `Y`.
`labelno`	The universally denying label, `N`.
`labeltop`	The top lattice value, `ffff...`.
`labelbot`	The bottom lattice value, `0000...`.

NAME

labeq, lable, labmax, labmin – compare security labels

SYNOPSIS

```
#include <sys/label.h>

labEQ(x, y)
struct label *x, *y;

labLE(x, y)
struct label *x, *y;

struct label labMAX(x, y)
struct label *x, *y;

struct label labMIN(x, y)
struct label *x, *y;
```

DESCRIPTION

LabEQ returns 1 if *x* and *y* point to equal labels, otherwise 0. The result is 1 if and only if neither argument is 0, the flag fields are the same, and, when the flag fields are `L_BITS`, the lattice values are the same.

LabLE returns 1 if the security label pointed to by *x* compares less than or equal to the security label pointed to by *y*. An improper argument is treated as if it had flag `L_NO`. If one of the labels has flag `L_YES`, the result is 1; otherwise if one of the labels has flag `L_NO`, the result is 0; otherwise the the result is 1 if and only if the lattice value of *x* is bitwise less than or equal to the lattice value of *y*. (Inequalities involving `L_YES` and `L_NO` are not transitive.)

LabMAX and *labMIN* respectively return the maximum (bitwise OR) and minimum (bitwise AND) of lattice values of labels pointed to by *x* and *y*. An improper argument is treated as if it had flag `L_NO`. If one of the labels has flag `L_YES`, the result is the other label; otherwise if one of the labels has flag `L_NO`, the result has flag `L_NO`.

The privilege and frozen-label fields of the labels are disregarded by all of these functions.

SEE ALSO

getflab(2)

NAME

labtoa, atolab, atopriv, privtoa – security label conversion

SYNOPSIS

```
#include <sys/label.h>

char *labtoa(labp) struct label *labp;
struct label *atolab(string) char *string;

atopriv(string) char *string;

char *privtoa(n)
```

DESCRIPTION

Labtoa returns a pointer to a null-terminated ASCII string that represents the value of the security label pointed to by *labp*. The string has a form exemplified by

```
guxnlp guxnlpFY 0000 0000 ...
```

The characters of the first group `guxnlp` denote capabilities `T_LOG`, `T_UAREA`, `T_EXTERN`, `T_NOCHK`, `T_SETLIC`, and `T_SETPRIV` respectively. Characters of the second group denote corresponding licenses; see *getplab*(2). Missing capabilities or licenses are denoted by `-`.

The character shown as `F` denotes the fixity of the label. It may be a space (loose), `F` (frozen), `R` (rigid), or `C` (constant) The character shown as `Y` denotes the label's flag. It may be a space for a lattice label, `N` for `L_NO`, `Y` for `L_YES`, or `U` for the erroneous flag value 0.

Each group of four zeros may be any four lower case hex digits representing the value of two bytes of the lattice value. Repeating groups at the end of the string are denoted `...`.

Atolab inverts the process. The order of characters in, and length of, privilege strings are arbitrary, except that a nonempty license string must be preceded by a nonempty capability string. The order of characters from the set `YNUFRC` is arbitrary. Spaces must separate nonempty capability and license strings, and may be interspersed arbitrarily after the license string. A final `...` causes the last four hex digits to be repeated, provided the preceding label contains a multiple of four digits. A short or missing lattice value is padded with zeros.

Atopriv converts a string of characters from the set `guxnlp-` into privilege bits that may be stored in the `lb_t` or `lb_u` fields of a label structure. The order and number of characters are arbitrary.

Privtoa is inverse to *atopriv*.

SEE ALSO

getflab(2), *getplab*(2), *getlab*(1)

DIAGNOSTICS

Atolab returns 0 for unrecognizable input.

Atopriv returns the negative value `~(T_LOG|T_UAREA|T_EXTERN|T_NOCHK|T_SETLIC|T_SETPRIV)` for unrecognizable input.

BUGS

The value returned by *labtoa*, *atolab*, or *privtoa* points to a static buffer that is overwritten at each call.

NAME

xs, enroll, verify, reverify, keynotary – certification functions

SYNOPSIS

```
char *xs(char *key, char *buf, int n)
enroll(char *name, char *oldkey, char *newkey)
verify(char *name, char *xsum, char *buf, int n)
rverify(char *name, char *xsum, char *buf, int n)
keynotary(char *key1, char *key2)
```

DESCRIPTION

All these funcitons except *xs* must be linked with option `-lipc` of *ld*(1).

Xs composes a cryptographic checksum of the *n* characters starting at *buf.* The *key* argument points to an 8-character checksumming key. A pointer is returned to a null-terminated ASCII checksum.

Enroll registers a checksumming key for user *name* with *notary*(1), only one checksumming key per user name at a time. On first registry the *oldkey* argument is ignored. On subsequent registries, the *oldkey* argument must match the currently stored checksumming key. The new checksumming key is *newkey*; if *newkey* is trivial, *name* is deregistered.

Verify consults the notary oracle to check the validity of a checksum composed by *xs.* A non zero return value signifies that the checksum was calculated using the checksumming key registered with the notary oracle as belonging to user *name*. *Rverify* does what *verify* does, but leaves the connection to the oracle open until presented with a NULL value for *name*. Hence, subsequent calls to *rverify* should be quicker.

Keynotary is used to tell the notary daemon the key for its private encrypted data. *Key1* is the key the data is currently encrypted with; *key2* (if nonzero) is the key to use in the future. A file descriptor is returned, from which diagnosic information may be read.

SEE ALSO

notary(1), *ipc*(3)

DIAGNOSTICS

Verify and *enroll* return zero on failure, otherwise nonzero.

NAME

pex, unpex – obtain process-exclusive file access

SYNOPSIS

```
#include <sys/pex.h>

int pex(fd, seconds, pexbuf)
struct pexclude *pexbuf;

int unpex(fd, seconds)
```

DESCRIPTION

Pex tries, using the ioctl call, to obtain exclusive access to the file designated by file descriptor *fd*; see *pex*(4). If *pexbuf* is nonzero, facts about the other end of the pipe are placed in the object *pexbuf* points to, as described in *pex*(4).

If *fd* refers to a stream, *pex* first empties the input and output queues, flushing if *seconds* is negative, and otherwise waiting up to the specified time interval for the queues to drain. If the queues do not drain, an error results.

Unpex uses to try to reverse the effect of *pex*, again flushing or draining queues as specified by *seconds.*

On a pipe, *pex* or *unpex* succeeds only if the process at the other end answers with an FIOPX or FIONPX ioctl respectively. *Pex* and *unpex* should not be used to answer.

SEE ALSO

pex(4)

DIAGNOSTICS

Pex returns –1 on failure, 0 on success, and 1 for a half-pexed pipe.

NAME

pwquery, pexpw – password services

SYNOPSIS

```
pwquery(fd, name, param)
char *name;
char *param;

char *pexpw(fd, prompt)
char *prompt;
```

DESCRIPTION

Pwquery calls upon the password server, *pwserv*(8) to cause a password to be demanded from file descriptor *fd* and checked against the password for the named user. It is loaded by option `-lipc` of *ld*(1).

Echoing is disabled during the transaction, and the server is persnickety about when to use an Atalla challenge/response dialogue and when to use *crypt*(3)-style passwords. A negative return value indicates a protocol error in reaching the server or that the server is not trusted. A zero return value indicates rejection of the password. A positive return value indicates approval of the password.

The argument *param* may be zero (for vanilla password service) or may point to a blank-separated list of one or more keywords. Currently only one keyword is understood:

`pex` Reject the password if the stream is unpexable.

Pexpw reads a password from the indicated file descriptor, after prompting with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most 8 characters. The dialogue is not attempted if it cannot be protected from eavesdropping by the process-exclusive mechanism of *pex*(4).

FILES

`/cs/pw`
`/etc/pwserv`

SEE ALSO

ipc(3), *getpass*(3), *pex*(4) *pwserv*(8)

BUGS

Pexpw returns a pointer to static memory that is overwritten at every call.

NAME

proc, stream – changes to manual

DESCRIPTION

This section covers small changes in the named manual pages for IX relative to v10.

proc

The groupid of all files in `/proc` is –1. No process can have this groupid.

stream

Changed *ioctl*(2) calls for streams:

`TIOCSPGRP`
: The process group can be set only to the current process id unless the process has capability `T_UAREA`.

`FIORCVFD`
: Deliver a structure pointed to by *param*:

```
struct passfd {
        int   fd;
        short uid;
        short gid;
        short nice;
        char  logname[8];
        char  cap;
};
```

The call blocks until there is something in the stream. If data is present, it returns `EIO`. If a file descriptor has been sent from the other end of the pipe by `FIOSNDFD`, `FIORCVFD` fills in the user and group ID of the sending process, its niceness (see *nice*(2)), its login name, its capabilities in the form of the field `lb_t` (see *getflab*(2)), and a file descriptor for the file being sent; the file is now open in the receiving process.

New *ioctl* calls:

`FIOGSRC`
: Copy the stream identifier to the `SSRCSIZ`-byte string pointed to by *param.*

`FIOSSRC`
: Copy the `SSRCSIZ`-byte string pointed to by *param* into the stream identifier. Capability `T_EXTERN` is required; see *getplab*(2). A newly created stream has an empty stream identifier. It is customary to set the stream identifier on network connections to identify the source. Successful password demands may also be recorded in the stream identifier for the benefit of *pwserv*(8).

SEE ALSO

session(1), *src*(5)

NAME

log – security log file

SYNOPSIS

`#include <sys/log.h>`

DESCRIPTION

The special files `/dev/log/log00` through `/dev/log/log15` refer to ‘repository’ files nominated by *syslog*(2).

The kernel automatically records selected events on the ‘system log file’ `/dev/log/log00` in the form described in *log*(5).

Any process with write access may write on a log file; no process has read access. Each write places in the repository file a *log*(5) record with `code` = `LOG_USER`. The data written are truncated to `LOGLEN` bytes and placed in the `body` field. When logging is not turned on, a log file acts like a write-only `/dev/null`.

FILES

`/dev/log/*`

SEE ALSO

syslog(2), *log*(5), *syslog*(8)

NAME

pex – ioctl requests for process-exclusive access

SYNOPSIS

```
#include <sys/pex.h>

ioctl(fildes, FIOPX, p)
struct pexclude *p;

ioctl(fildes, FIONPX, p)
struct pexclude *p;

ioctl(fildes, FIOQX, p)
struct pexclude *p;

ioctl(fildes, FIOAPX, p)
struct pexclude *p;

ioctl(fildes, FIOANPX, p)
struct pexclude *p;
```

DESCRIPTION

These *ioctl*(2) requests provide and check temporary exclusive access to an input/output source. `FIOPX` marks as ‘pexed’ the file or pipe end referred to by *fildes.* On a pexed file *read, write*(2), and most forms of *ioctl* work only in the pexing process. Moreover, these operations do not work in any process on a half-pexed pipe (a pipe with exactly one pexed end). The mark remains until the pexing process requests `FIONPX` or closes all file descriptors that refer to the file.

When *fildes* refers to a stream, `FIOPX` and `FIONPX` require the stream’s input and output queues to be empty; *pex*(3) gives a method for emptying them. When *fildes* refers to a pipe, the far end of which is unpexed, `FIOPX` waits, with timeout, for an answering `FIOPX` or `FIONPX` at the far end. `FIONPX` waits similarly when the far end is pexed. Either request returns 1 when it leaves a pipe with exactly one end pexed. A pipe must cycle through the fully unpexed state between fully pexed states; from the time one end becomes unpexed until the far end does too, `FIOPX` on the unpexed end will return error `ECONC`.

If argument *p* is nonzero, the structure it points to is filled in with information about the pexedness of the file and about the process at the far end of a pexed pipe. The format, defined in `<sys/filio.h>` is:

```
struct pexclude {
      int oldnear;      /* FIOPX or FIONPX: state at begining of call */
      int newnear;      /* FIOPX or FIONPX: state at end of call */
      int farpid; /* -1 if not pipe, 0 if not pexed, else process id */
      int farcap; /* if farpid>0, capabilities */
      int faruid; /* if farpid>0, user id */
};
```

Capabilities are represented as in the `lb_t` field of a label; see *getflab*(2).

`FIOQX` obtains the information without affecting state.

Read, write, or *ioctl* calls that fail due to pexedness return error `ECONC`. The only *ioctl* requests that may succeed on a half-pexed pipe are `FIOCLEX`, `FIONCLEX`, `FIOPX`, `FIONPX`, and `FIOQX`. A half-pexed pipe is deemed ready by *select*(2).

`FIOANPX` and `FIOAPX` modify the response of open stream device files to `FIOPX` requests. They require `T_EXTERN` capability; see *getplab*(2). After `FIOANPX` all `FIOPX` requests on the special file return 1 and leave the device in an unusable state (as if the device driver were a process at the far end of a pipe, always responding `FIONPX`). The treatment is reversed with `FIOAPX`. This mechanism allows a terminal to be denounced to the kernel as being attached to an untrusted remote computer that cannot guarantee the exclusivity asked by `FIOPX`.

EXAMPLES

A program collecting a password wishes to exclude other programs from the dialogue. The following code does the trick. (When the dialogue passes through *mux*(9.1) or *con*(1), downstream stages of the path to the terminal can be assumed to be similarly pexed, provided `FIOPX` succeeds.)

```
#define ok(p) (p->farpid==-1 || p->farpid>0 && p->farcap!=0)
struct pexclude x;
if(ioctl(fd, FIOPX, &x) == 0 && ok(&x)) {
      static char buf[9];
      write(fd, promptstr, strlen(promptstr));
      read(fd, buf, 8);
      s = buf;
} else
      s = 0;
ioctl(fd, x.oldnear, 0);      /* restore state */
```

An intervening trusted program, with a policy of recognizing exclusive access only for trusted processes, may cooperate with

```
n = read(fd, buf, BUFSIZE);
if(n == -1 && errno == ECONC) {
      if(ioctl(fd, FIOPX, &pexcode)!=0 || pexcode.farcap==0)
            ioctl(fd, FIONPX, 0);
      } else  /* improper pexing */
```

SEE ALSO

ioctl(2), *pipe*(2), *stream*(4), *pex*(3)

DIAGNOSTICS

`EBADF`, `ECONC`, `EFAULT`, `EIO`, `ENOTTY` (`FIOAPX` and `FIOANPX`)
`ECONC` for forbidden IO calls in other processes.
`EBUSY` for an undrained queue.

NAME

filsys, fstab, passwd – changes to manual

DESCRIPTION

This section covers small changes in the named manual pages for IX relative to v10.

filsys

A disk inode in a regular file system contains an extra field for the file's security label.

```
#include <sys/label.h>

struct label labeldi;
```

fstab

The table of normally mounted file systems, `/etc/fstab`, contains an extra field for the file system ceiling; see *fmount*(2).

passwd

In addition to the usual password file, `/etc/passwd`, there is a highly secret file `/etc/pwfile`, which is used by *pwserv*(8) to authorize clearances. Each content line contains the following fields, separated by colons:

name
encrypted password
SNK key
process license (unused)
clearance (maximum ceiling)

The license and label fields are in the form understood by *labtoa*(3); thus the label field may contain white space. Lines with fewer than five fields are ignored.

The name field contains a user name for option `-u` of `/bin/session`. It is customary, but not necessary, for names in *pwfile* also to be registered in *passwd*(5).

The SNK field gives a 24-digit octal key for a Secure Net Key (or Atalla) challenge box.

The label field gives the maximum permissible label for option `-l`, and the ceiling label otherwise.

NAME

log – format of security logging records

SYNOPSIS

`#include <sys/log.h>`

DESCRIPTION

The structure of system log file records as declared in `<sys/log.h>` is

```
struct logbuf {
        short  len;          /* total length of whole record */
        short  pid;          /* process id */
        long   slug;         /* transaction number */
        char   code;         /* kind of record */
        char   mode;         /* sub-kind */
        char   colon;        /* ':', aids sync */
        char   body[LOGLEN];
};
```

The `code` field identifies the kind of record; for legal values see the include file. In kernel records the `mode` field identifies where in the kernel the logging record originated, for user records it contains the minor device number of the `/dev/log/log`*xx* file used to create the record.

The `body` field contains the logging record proper; its actual length is determined from the `len` field. In kernel records the `body` is a sequence of values, each prefixed by one or more format bytes according to the following list. Multibyte numbers are represented low byte first.

`s` Next two bytes are a byte count for following string.

`$` Next one byte is a byte count for following string, which is typically a file component name.

`C` Next byte is a byte count for following string, which is the command name.

`j` Next value is a security label: two bytes of `lb_priv` followed by two bytes of index into the kernel's shared label table for the lattice value of the label; see *getflab*(2).

`J` Next value is a security label: two bytes of `lb_priv` followed by two bytes of index into the kernel's shared label table for the lattice value of the label, followed by 60 bytes of bits of the lattice value of the label.

n Next *n* bytes (n *=1,2,3,4)* represent a number.

`I` Next bytes name an inode: two bytes of device followed by two bytes of inumber.

`E` The current system call suffered an `ELAB` error.

`e` Next byte is an *errno* code; see *intro*(2).

The various bits of the log mask (see *syslog*(2) are named `LN`, `LS`, `LU`, `LI`, `LD`, `LP`, `LL`, `LA`, `LX`, `LE`, `LT`, with the same meanings as the corresponding key letters defined in *syslog*(8).

FILES

`/dev/log`

SEE ALSO

syslog(2), *log*(4), *syslog*(8)

BUGS

The various kinds of kernel logging records are understandable only by reading the kernel source code.
It takes 7 bytes, not 4, to name an inode.

NAME

privs – privilege file

DESCRIPTION

The file `/etc/privs` expresses the rules whereby *priv*(1) grants privilege. It consists of a list of statements, each terminated by a semicolon. One or more comments, each extending from # to newline, may precede each statement.

Rights

Rights are defined thus:

`DEFINE` *rights-list* `;`

Each right in the comma-separated rights-list has a name, and optionally a parenthesized parameter type. The types are

`LAB` Label, ordered by lattice value.

`RE` Regular expression ordered by language inclusion. Regular expressions are in the form of *regexp*(3), with enclosing `^(` and `)$` understood.

`PRIV` Set of privileges in *atopriv* form, ordered by inclusion; see *labtoa*(3).

Examples:

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

Rights are identifiers used solely by *priv*; they have no other manifestation in the system. In the example, the `ceiling` right involves label comparisons, but has no necessary connection to process ceilings. The name could be changed globally to, say, `floor` without affecting the interpretation of `/etc/privs`.

Authorization

Authorization is expressed by a tree. Nodes of the authorization tree are named, like files in the file system, by full pathnames starting from the root, `/`. Associated with each node are statements to grant rights, and statements to admit access to the node. Rights are monotone in the tree: the rights at a node must be a subset of the rights at its parent. Access to a node implies access to its children.

Right-granting statements have the form

`RIGHTS` *nodename rights-list* `;`

A *rights-list* is as in a rights definition, but with explicit values for parameters. White space or one of the metacharacters `;,()` may be included in a value by placing double quotes around it. Examples:

```
RIGHTS /admin            priv(upxnl), ceiling(ffff...);
RIGHTS /admin/security   priv(p), ceiling("ffff ...");
RIGHTS /admin/operations priv(xn)
```

Access statements have the form

`ACCESS` *nodename pred-list* `;`

Access to the named node is granted when the comma-separated *pred-list* is nonempty and all the predicates in the list are satisfied. A node may have more than one `ACCESS` statement. Legal predicates are

`ID(` *lognames* `)`
A regular expression for login names that have access to this node.

`PW(` *name ...* `)`
The password associated with one of the *names* in *pwfile*(5) must be presented.

`SRC(` *source* `)`
A regular expression for the stream identifier of the standard input.

Rules

Rules give patterns for requests and show the prerequisite rights for and the actions to carry out each request:

`REQUEST(` *arguments* `) NEEDS` *rights* `DOES` *actions* `;`

The request part shows *arguments* supplied to *priv*(1); normally the arguments spell out the prefix of a UNIX command. The `NEEDS` part tells what rights are needed to perform the request. The rights are as in a rights statement, with substituted parameters; see ‘Parameter values’.

If the process has access to a node that grants the needed rights (with the parameter in each grant dominating the parameter of the corresponding need), then the *actions* for the request are performed. Otherwise the request is denied. Legal actions are

`PRIV(gunxlp)`
: Set one or more process licenses, abbreviated as in *labtoa*(3).

`EXEC(` *args* `)`
: Execute a program given by the *args.* Members of the *args* list are separated by white space and may specify substitutions; see ‘Parameter values’. `EXEC` does not do a *sh*(1)-like `$PATH` search.

`DAEMON(` *args* `)`
: Same as `EXEC`, but do not wait for the command to complete.

`CEILING(` *label* `)`
: Set the process ceiling.

`PRIVEDIT(` *node file* `)`
: Read editing commands from the named *file*. Only the subtree at *node* is editable; nodes closer to the root cannot be touched.

`ANYSRC`
: Skip the normal check for a trusted source; see *priv*(1).

The order in which nodes of the authorization tree are visited in evaluating a `NEEDS` clause is undefined, however at each node the predicates of the request are evaluated in order. The actions of a granted request are also performed in order, with effects such as privilege settings persisting until the end of the *priv* command or until overridden by a later action.

Parameter values

Parameter values appear in members of `NEEDS` and `DOES` lists. A value may be surrounded by double quotes, in which case the value may contain white space or one of the metacharacters `,;()`. A value may contain substitution marks, `$0`, `$1`, ... Each such mark is replaced from the *priv* invocation, `$0` standing for the match to the first *argument* of the `REQUEST` and so on. If a star is appended to the mark (e.g. `$0*`, `$1*`), the argument and all following ones are copied into the parameter list. Nothing can follow a star mark in a parameter.

Editing

Statements of the above forms may be used with action `PRIVEDIT` to augment a *privs* file. Further types of statements exist for editing only:

`RMDEFINE` *rights-list* `;`
: Remove all occurences of the listed rights from the file.

`RMACCESS` *nodename pred-list* `;`
`RMRIGHTS` *nodename rights-list* `;`
: Remove the given access list or the given rights from the named node. If the list is empty, remove all access lists or rights.

`RMREQUEST(` *arguments* `)` *;*
: Remove the `REQUEST` with identical *arguments.*

`RMNODE` *path-list* `;`
: Remove the listed subtrees.

`DEFINE`, `RMDEFINE`, `REQUEST`, and `RMREQUEST` are understood to modify the root.

EXAMPLES

```
REQUEST(session -l)
      NEEDS ceiling($2)
      DOES PRIV(nx) EXEC(/bin/session -l $2);
REQUEST(/etc/downgrade -l)
      NEEDS downgrade($2)
      DOES PRIV(nx) EXEC($*);
```

FILES

`/etc/privs`

SEE ALSO

priv(1), *privserv*(8)

BUGS

There is no way to quote a newline or an initial double quote in parameters.

If an `ACCESS` or `RMACCESS` statement contains duplicate predicates, `RMACCESS` may remove an unintended list.

NAME

src – form of a stream identifier

DESCRIPTION

Stream identifiers, defined in *stream*(4), are conventionally set by *init*(8) and *dkmgr*(8) to designate the source of the login stream. A datakit source begins with `dk!` followed by a dial string.

Session(1) may append to the stream identifier of the standard input a colon and a name, which is understood by *pwserv*(8) as an assertion that the agent on that stream knows the password associated with that name, which obviates further demands for that password.

EXAMPLES

```
dk!201/mu/attbl:doug
```

SEE ALSO

getstsrc(3)

NAME

apx – mark an open stream device trusted

SYNOPSIS

`/etc/apx` [*arg*]

DESCRIPTION

By default, a freshly opened stream device has the `APX` bit cleared: it will reject all pex requests. If invoked without an argument, *apx* will set the APX bit on its standard input (by calling the `FIOAPX` control). If invoked without an argument the `APX` bit is cleared. *Apx* needs licence `T_EXTERN` to run. It is usually automatically invoked at login time, provided that the source identifier of the standard input of the login session is worthy.

FILES

`/etc/privs`

SEE ALSO

pex(4)

NAME

init, mount – changes to manual

DESCRIPTION

This section covers small changes in the named manual pages for IX relative to v10.

init

In single-user operation, *init* invokes *nosh*(8) with security label set at bottom and all capabilities; see *getplab*(2). At the end of single-user operation, *init* invokes *nosh* to run the startup script `/etc/rs.nosh`, again with bottom label and all capabilities.

In multiuser operation, *init* opens each terminal port with a security label set to the a 'floor' value, which is the label of the file `/etc/floor`.

mount

New option:

`-l` *label*

The *label,* specified as in *labtoa*(3), becomes the file system ceiling; see *fmount*(2).

NAME

cl, integrity – file system label check

SYNOPSIS

`/etc/cl` [*specfile* | *dir*] ...

`/etc/integrity` [*rootdir*]

DESCRIPTION

Cl examines file trees for correctness of labels. Each *specfile* argument names a file containing a description of the labels expected in a given subtree of a file system. Each line of a *specfile* has the form

```
filename uid,gid mode capabilities licenses label
```

User and group ids are specified in the style of *chown*(8). The mode is specified in the style of *chmod*(2); only the 07777 bits are significant. Capabilities and licenses are in the style of *atopriv*; see *labtoa*(3). The label is in the style of *atolab,* without capabilities or licenses.

The first valid line names the root of the tree in question. Subsequent lines name particular files in the tree. A report is made for each 'suspicious' file and for each particular file which does not match its description in *specfile*.

A suspicious file is a file that is not named in the *specfile* for which one of the following holds:

The label has flag `L_UNDEF` or `L_YES`.
The file is a special file the label flag is `L_NO`.
The file is not a special file the label flag is not `L_NO`.
The lattice value of the label is not dominated by the label in the first line of *specfile*.
The capability or license is not dominated by the corresponding value in the first line of *specfile*.

Each named directory argument *dir* is treated as if there were a *specfile* argument consisting of just a single line

dir `bin,bin 666 ----- ----- 0000...`

Integrity surveys the directory tree dependent from *rootdir*, or `/` if no *rootdir* is given. It reports non-bottom labels, which are possible signs of loss of integrity – modification without privilege. The search cuts off at directories with non-bottom labels.

SEE ALSO

getflab(2), *ftw*(3), *lcheck*(8)

BUGS

Extraneous diagnostics may be produced if this command is applied to active file systems.

NAME

nosh – 'no-suprise' shell, a sub-standard command interpreter

SYNOPSIS

`/etc/nosh` [*file*]

`priv nosh -gunxlp` *file*

DESCRIPTION

Nosh executes commands read from its standard input or from the named *file*. It has few of the advanced features of *sh*(1), making it more trustable for use in security administration tasks. In the second usage, *nosh* is endowed with one or more of the licenses `gunxlp`; see *labtoa*(3).

Commands

A command is either *simple* or *builtin.* Each command consists of a sequence of *words* separated by white space, terminated by a new-line character or end of input. Backslash quoting and sharp commenting are honored. The first word specifies the name of the command to be executed. If the command name matches one of the builtins listed below it is executed in the shell process. If the command name matches no builtin command, it is taken to be the pathname of an executable file; the name must begin with `/` or `.`. A new process is created and an attempt is made to execute the file via *exec*(2) with an empty environment.

Input-Output Redirection

The standard input is inherited by simple commands. Simple `>` output redirection to named files as in *sh*(1) works only for simple commands, and only for file descriptors 1 (default) and 2.

Builtin Commands

`cd` *dir* Change the current directory to *dir.*

`exit` *status*
: Exit with given status, 0 by default.

`set +e`
`set -e`
: Turn an ignore-error switch on (`+e`, default) or off (`-e`). *Nosh* normally ignores nonzero exit status from an executed command, but exits with that status if `-e` is set.

`set +x`
`set -x`
: Refrain from echoing (`+x`, default) or echo (`-x`) each command as it is executed.

`lmask` *licenses command* [*arg* ...]
: Run a simple command, allowing licenses indicated by a nonempty string from the set `gunxlp-` to be inherited from *nosh.* Normally no licenses are inherited.

Missing features

Features of *sh*(1) that *nosh* lacks include: background commands, pipelines, compound commands, most builtins, multicharacter quotation, command substitution, parameter substitution, variables, environments, file name generation, redirection of input, signal traps, search paths, mail notification, `.profile`, user specification of prompts.

DIAGNOSTICS

Nosh prints nonzero exit or termination status of executed commands as octal numbers labeled `e=` and `t=`; see *wait*(2). If invoked with a *file* argument, it exits unconditionally for nonzero termination status or syntax error, and conditionally (under control of `set`) for nonzero exit status.

Nosh exits immediately if invoked with more than one argument, if invoked with an argument with a relative path name, if invoked by a relative path name, or if invoked with interrupt or quit signals ignored.

SEE ALSO

sh(1)

NAME

privserv – privilege server

SYNOPSIS

`lmask nuxl /etc/privserv` [*option* ...]

DESCRIPTION

Privserv is the keeper and interpreter of the *privs*(5) file. *Priv (1)* calls on *privserv* to hand out privileges in accordance with the rules given in *privs*. *Privserv* is a permanent process, normally started by the boot script *rc*(8). It receives service requests through the mounted pipe `/cs/priv`. The options are

`-p` *name*
: The file name of the server, `/etc/privserv` by default (used to reinvoke the priv server when the *privs*(5) file is modified by a `PRIVEDIT` request.)

`-m` *mountpt*
: The file system mount point for privilege service, `/cs/priv` by default.

`-l` *logfile*
: The file in which to record logging information, `/usr/adm/privlog` by default.

`-f` *privs*
: The data base of privileges, `/etc/privs` by default. Unless *privs* is itself a privileged file, *privserv* will not actually grant the privileges there specified.

FILES

`/etc/privs`
`/cs/priv`

SEE ALSO

priv(1)

NAME

pwserv – password verification service

SYNOPSIS

`/etc/pwserv`

DESCRIPTION

Pwserv, normally started from *rc*(8), handles password verification requests initiated by (say) *pwquery*(3) through the conventional process mount point `/cs/pw`. When a request is made a file descriptor (called the 'line' below) is passed to *pwserv* together with a user name and an optional parameter string. Normally, *pwserv* writes a prompt on the line, reads a reply, and returns an indication of success to the invoking client. Valid passwords are taken from the file `/etc/pwfile`, which lists for each user an ordinary (encrypted, *crypt*(3)-style) password and an SNK (Secure Net Key) challenge-response key. Before prompting, an `FIOPX` IO control is attempted to render the line to the end user private; see *pex*(4). If this succeeds either a classical or an Atalla password is accepted. If the pex bid fails, the prompt warns that the line is not private, and only an SNK response is accepted.

In the pexed case the prompt looks like `Password(pjw:31416):` and in the unpexed case like `Password(TAPPED LINE:01492):` The five digit string after the colon is the Atalla challenge string. Only the first five digits of the Atalla response string are significant. Hex digits in the response must be typed in lower case.

Possible values of the optional parameter string are

`pex` (specified by opening the server with `ipcopen("/cs/pw!pex")`) Accept passwords only if the `FIOPX` succeeds.

When the line's stream identifier asserts previous confirmation of the same password, *pwserv* answers affirmatively without demanding a password; see *session*(1) and *src*(5).

FILES

`/etc/pwserv`
`/etc/pwfile`

SEE ALSO

pwquery(3), *ipc*(3), *pex*(4), *stream*(4), *pwfile*(5), *passwd*(1)

BUGS

Jammable.

NAME

syslog, logpr – system security logging

SYNOPSIS

`priv syslog` *command* [*arg2* [*arg3*]]

`/etc/logpr` *file* [*offset*]

DESCRIPTION

Syslog controls the mandatory logging scheme. License `T_LOG` is required. The variety of different commands and command formats reflects the full complexity of the protean *syslog*(2) system call. In the usages given below a *mask* argument is a combination of letters `NILESDATUPX`, meaning:

- `N` Record all uses of file names.
- `S` Record all seek calls.
- `U` Record all writes to the ‘u area’.
- `I` Record all accesses of inode contents.
- `D` Record possession and use of file descriptors.
- `P` Record process history: *exec*(2), *fork*(2), *kill*(2), *exit*(2).
- `L` Record all explicit changes of labels by *setflab* (see *getflab*(2)) and *setplab* (see *getplab*(2)).
- `A` Record all changes of labels.
- `X` Record all uses of privilege.
- `E` Record all `ELAB` error returns.
- `T` Record all uses of a traced file or process.

Valid arguments to *syslog* are:

`on` *file logdev*
: Nominate *file* as repository for user generated logging records written to logging special file *logdev*. *File* must be a full path name, and must be openable for writing. If *logdev*’s minor device number is zero, *file* will also receive mandatory (kernel generated) logging records. *Logdev* may be a full path name or a minor device number.

`off` *logdev*
: Cancel the effect of an `on` command.

`get` *n*
: Print the value of the *n*-th log mask. Values of *n* are 0, 1, 2, or 3 for the ‘poison’ masks; 4 is ‘global’ mask.

`set` *n mask*
: Set the value of the *n*-th log mask.

`fget` *file*
: Print the poison level of *file*, one of the integers 0, 1, 2, or 3. *File* must be the full path name of a readable file.

`fset` *file n*
: Set the poison level of *file* to *n*. *File* must be the full path name of a readable file.

`pget` *pid*
: Print the logging mask of process *pid*.

`pset` *pid mask*
: Set the logging mask of process *pid* to *mask*.

Logpr converts to cryptic ASCII the cryptic binary format of a log file described in *log*(5). The optional numerical byte offset tells where in the file printing is to start.

FILES

`/dev/log/log00` where *syslog* makes voluntary entries

SEE ALSO

syslog(2), *log*(4), *log*(5).

DIAGNOSTICS

‘Covert channel warning’: the log file has a label that is neither top nor flagged `L_NO`.

BUGS

Logpr is very primitive.

NAME

xs – checksums

SYNOPSIS

`xs` [-s] [-k *keystring*] [-f *official-list*] *file* ...

DESCRIPTION

Xs computes and reports checksums of named files, one report per line, in the form

filename s1 s2 s3 s4

where the checksum comprises four groups of four hex digits each. The checksum algorithm may be perturbed by specifying a *keystring* argument. The `-s` argument causes the file's mode, label, owner and group to enter into the checksum calculation.

The `-f` argument causes *xs* to verify checksums of files against values given in the *official-list* file, which has the format of the output of an earlier *xs* run: lines consisting of one file name followed by four groups of hex digits per line. Text after a # sign is ignored.

The checksum algorithm used is meant to be secure: to create a file whose checksum agrees with that of another given file is very difficult.

EXAMPLES

```
xs -s `find /bin -print` | xs /dev/stdin
```

This should return a different value if `/bin` changes in any way.