NAME

cap_get_proc, cap_set_proc, capgetp, cap_get_bound, cap_drop_bound, cap_get_ambient, cap_set_ambient, cap_reset_ambient, cap_get_secbits, cap_set_secbits, cap_get_mode, cap_set_mode, cap_mode_name, cap_get_pid, cap_setuid, cap_prctl, cap_prctlw, cap_setgroups − capability manipulation on processes

SYNOPSIS

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

cap_t cap_get_proc(void);
int cap_set_proc(cap_t cap_p);

int cap_get_bound(cap_value_t cap);
CAP_IS_SUPPORTED(cap_value_t cap);

int cap_drop_bound(cap_value_t cap);
int cap_get_ambient(cap_value_t cap);
int cap_set_ambient(cap_value_t cap, cap_flag_value_t value);
int cap_reset_ambient(void);
CAP_AMBIENT_SUPPORTED();

unsigned cap_get_secbits(void);
int cap_set_secbits(unsigned bits);
cap_mode_t cap_get_mode(void);
const char *cap_mode_name(cap_mode_t mode);
int cap_prctl(long int pr_cmd, long int arg1, long int arg2,
              long int arg3, long int arg4, long int arg5);
int cap_prctlw(long int pr_cmd, long int arg1, long int arg2,
               long int arg3, long int arg4, long int arg5);
int cap_set_mode(cap_mode_t mode);

#include <sys/types.h>

cap_t cap_get_pid(pid_t pid);
int cap_setuid(uid_t uid);
int cap_setgroups(gid_t gid, size_t ngroups, const gid_t groups);
```

Link with *−lcap*.

DESCRIPTION

cap_get_proc() allocates a capability state in working storage, sets its state to that of the calling process, and returns a pointer to this newly created capability state. The caller should free any releasable memory, when the capability state in working storage is no longer required, by calling **cap_free**() with the *cap_t* as an argument.

cap_set_proc() sets the values for all capability flags for all capabilities to the capability state identified by *cap_p*. The new capability state of the process will be completely determined by the contents of *cap_p* upon successful return from this function. If any flag in *cap_p* is set for any capability not currently permitted for the calling process, the function will fail, and the capability state of the process will remain unchanged.

cap_get_pid() returns a *cap_t*, see **cap_init**(3), with the process capabilities of the process known to the caller as *pid*. If*pid* is 0, then the calling process' s capabilities are returned. This information can also be obtained from the */proc/<pid>/status* file. (The entries in that file can be translated with the **capsh −−decode=***XXX* command line.) When the caller is operating within a (**CLONE_NEWPID**) namespace, the numerical *pid* argument is interpreted in the range of that namespace. As such, the caller's idea of the target *pid* may differ from that of the target process when they are operating in different pid namespaces. See **pid_namespaces**(7) for details. Further, the returned *cap_t* value holds the capabilities that the target *pid*

thinks it has. If the target is operating in a (**CLONE_NEWUSER**) namespace, the system wide privilege of those user namespace capabilities may be substantially reduced. See **user_namespaces**(7) for details.

cap_get_bound() with a *cap* as an argument returns the current value of this bounding set capability flag in effect for the calling process. This operation is unprivileged. Note, a macro function **CAP_IS_SUPPORTED(cap_value_t** cap**)** is provided that evaluates to true (1) if the system supports the specified capability, *cap*. If the system does not support the capability, this function returns 0. This macro works by testing for an error condition with **cap_get_bound**().

cap_drop_bound() can be used to lower the specified bounding set capability, **cap**. To complete successfully, the prevailing *effective* capability set must have a raised **CAP_SETPCAP**.

cap_get_ambient() returns the prevailing value of the specified ambient capability, or -1 if the capability is not supported by the running kernel. A macro**CAP_AMBIENT_SUPPOR TED**() uses this function to determine if ambient capabilities are supported by the kernel.

cap_set_ambient() sets the specified ambient capability to a specific value. To raise a specific ambient capability the *inheritable* and *permitted* flags of the calling process must contain the specified capability value. Raised ambient bits will only be retained as long as this remains true of the inheritable and permitted flags.

cap_reset_ambient() resets all of the ambient capabilities for the calling process to their lowered value. Note, the ambient set is intended to operate in a legacy environment where the application has limited awareness of capabilities in general. Executing a file, with associated filesystem capabilities, the kernel will implicitly reset the ambient set of the process. Further, changes to the inheritable set by the program code without explicitly fixing up the ambient set can also drop ambient bits.

cap_get_secbits() returns the securebits of the calling process. These bits affect the way in which the calling process implements things like setuid-root fixup and ambient capabilities.

cap_set_secbits() attempts to modify the securebits of the calling process. Note **CAP_SETPCAP** must be in the effective capability set for this to be effective. Some settings lock the sub-states of the securebits, so attempts to set values may be denied by the kernel even when the **CAP_SETPCAP** capability is raised.

To help manage the complexity of the securebits, libcap provides a combined securebit and capability set concept called a libcap mode. **cap_get_mode**() attempts to summarize the pre vailing security environment in the form of a numerical **cap_mode_t** value. A text representation of the mode can be obtained via the **cap_mode_name**() function. The vast majority of combinations of these values are not well defined in terms of a libcap mode, and for these states **cap_get_mode**() returns (**cap_mode_t**)0 which **cap_get_name**() identifies as *''UNCERTAIN''*. Supported modes are:**CAP_MODE_NOPRIV**, **CAP_MODE_HYBRID**, **CAP_MODE_PURE1E** and **CAP_MODE_PURE1E_INIT**.

cap_prctl() can be used to read state via the **prctl*()* system call.**

cap_prctlw() can be used to write state via the **prctl*()* system call.**

cap_set_mode() can be used to set the desired mode. The permitted capability **CAP_SETPCAP** is required for this function to succeed.

cap_setuid() is a convenience function for the **setuid**(2) system call. Where **cap_setuid**() arranges for the right effective capability to be raised in order to perform the system call, and also arranges to preserve the availability of permitted capabilities after the uid has changed. Following this call all effective capabilities are lowered.

cap_setgroups() is a convenience function for performing both **setgid**(2) and **setgroups**(2) calls in one call. The **cap_setgroups**() call raises the right effective capability for the duration of the call, and empties the effective capability set before returning.

RETURN VALUE

The functions **cap_get_proc**() and **cap_get_pid**() return a non-NULL value on success, and NULL on failure.

The function **cap_get_bound**() returns −1 if the requested capability is unknown, otherwise the return

value reflects the current state of that capability in the prevailing bounding set. Note, a macro function,

The all of the setting functions such as **cap_set_proc**() and **cap_drop_bound**() return zero for success, and −1 on failure.

On failure, *errno* is set to **EINVAL**, **EPERM**, or **ENOMEM**.

CONFORMING TO

cap_set_proc() and **cap_get_proc**() are specified in the withdrawn POSIX.1e draft specification. **cap_get_pid**() is a Linux extension.

NOTES

Neither glibc, nor the Linux kernel honors POSIX semantics for setting capabilities and securebits in the presence of pthreads. That is, changing capability sets, by default, only affect the running thread. To be meaningfully secure, however, the capability sets should be mirrored by all threads within a common program because threads are not memory isolated. As a workaround for this, **libcap** is packaged with a separate POSIX semantics system call library: **libpsx**. If your program uses POSIX threads, to achieve meaningful POSIX semantics capability manipulation, you should link your program with:

ld ... −lcap $(pkg-config −−libs −−cflags libpsx)

or,

gcc ... −lcap $(pkg-config −−libs −−cflags libpsx)

When linked this way, due to linker magic, libcap uses **psx_syscall**(3) and **psx_syscall6**(3) to perform state setting system calls. Notably, this also ensures that **cap_prctlw***()* can be used to ensure process control bits are shared over all threads of a single process.

capgetp() and capsetp()

The library also supports the deprecated functions:

int capgetp(pid_t *pid***, cap_t** *cap_d***);**

int capsetp(pid_t *pid***, cap_t** *cap_d***);**

capgetp() attempts to obtain the capabilities of some other process; storing the capabilities in a pre-allocated *cap_d*. See**cap_init**() for information on allocating an empty capability set. This function is deprecated; you should use **cap_get_pid**().

capsetp() attempts to set the capabilities of the calling process or of some other process(es), *pid*. Note that setting capabilities of another process is only possible on older kernels that do not provide VFS support for setting file capabilities. See **capset**(2) for information on which kernels provide such support.

If *pid* is positive it refers to a specific process; if it is zero, it refers to the calling process; −1 refers to all processes other than the calling process and process '1' (typically **init**(8)); other negative values refer to the −*pid* process group.

In order to use this function, the kernel must support it and the calling process must have **CAP_SETPCAP** raised in its Effective capability set. The capabilities set in the target process(es) are those contained in *cap_d*.

Kernels that support filesystem capabilities redefine the semantics of **CAP_SETPCAP** and on such systems, **capsetp**() will always fail for any target not equal to the calling process. **capsetp**() returns zero for success, and −1 on failure.

On kernels where it is (was) supported, **capsetp**() should be used with care. It existed, primarily, to overcome an early lack of support for capabilities in the filesystems supported by Linux. Note that on older kernels where **capsetp**() could be used to set the capabilities of another process, the only processes that had **CAP_SETPCAP** available to them by default were processes started as kernel threads. (Typically this includes **init**(8), kflushd and kswapd.) A kernel recompilation was needed to modify this default.

EXAMPLE

The code segment below raises the **CAP_FOWNER** and **CAP_SETFCAP** effective capabilities for the caller:

```
...
cap_t caps;
const cap_value_t cap_list[2] = {CAP_FOWNER, CAP_SETFCAP};

if (!CAP_IS_SUPPORTED(CAP_SETFCAP))
    /* handle error */

caps = cap_get_proc();
if (caps == NULL)
    /* handle error */;

if (cap_set_flag(caps, CAP_EFFECTIVE, 2, cap_list, CAP_SET) == -1)
    /* handle error */;

if (cap_set_proc(caps) == -1)
    /* handle error */;

if (cap_free(caps) == -1)
    /* handle error */;
...
```

Alternatively, to completely drop privilege in a program launched setuid-root but wanting to run as a specific user ID etc. in such a way that neither it, nor any of its children can acquire privilege again:

```
...
uid_t nobody = 65534;
const gid_t groups[] = {65534};

if (cap_setgroups(groups[0], 1, groups) != 0)
    /* handle error */;
if (cap_setuid(nobody) != 0)
    /* handle error */;

/*
 * privilege is still available here
 */

if (cap_set_mode(CAP_MODE_NOPRIV) != 0)
    /* handle error */
...
```

Note, the above sequence can be performed by the **capsh** tool as follows:

sudo capsh --user=nobody --mode=NOPRIV --print

where **--print** displays the resulting privilege state.

SEE ALSO

libcap(3), **libpsx**(3), **capsh**(1), **cap_clear**(3), **cap_copy_ext**(3), **cap_from_text**(3), **cap_get_file**(3), **cap_init**(3), **namespaces**(7), **pid_namespaces**(7), **user_namespaces**(7), **psx_syscall**(3), **capabilities**(7).