NAME

procctl - control processes

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/procctl.h>

int
procctl(*idtype_t idtype*, *id_t id*, *int cmd*, *void *data*);

DESCRIPTION

The **procctl**() system call provides for control over processes. The *idtype* and *id* arguments specify the set of processes to control. If multiple processes match the identifier, **procctl** will make a "best effort" to control as many of the selected processes as possible. An error is only returned if no selected processes successfully complete the request. The following identifier types are supported:

P_PID Control the process with the process ID *id*. *id* zero is a shortcut for the calling process ID.

P_PGID Control processes belonging to the process group with the ID *id*.

The control request to perform is specified by the *cmd* argument.

All status changing requests *_CTL require the caller to have the right to debug the target. All status query requests require the caller to have the right to observe the target.

The following commands are supported:

PROC_ASLR_CTL Controls the Address Space Layout Randomization (ASLR) in the program images created by execve(2) in the specified process or its descendants that did not changed the control nor modified it by other means. The *data* parameter must point to the integer variable holding one of the following values:

PROC_ASLR_FORCE_ENABLE Request that ASLR is enabled after execution, even if it is disabled system-wide. The image flag and set-uid might prevent ASLR enablement still.

PROC_ASLR_FORCE_DISABLE Request that ASLR is disabled after execution. Same notes as for PROC_ASLR_FORCE_ENABLE apply.

PROC_ASLR_NOFORCE Use the system-wide configured policy for ASLR.

PROC_ASLR_STATUS Returns the current status of ASLR enablement for the target process. The *data* parameter must point to the integer variable, where one of the following values is written:

PROC_ASLR_FORCE_ENABLE

PROC_ASLR_FORCE_DISABLE

PROC_ASLR_NOFORCE

If the currently executed image in the process itself has ASLR enabled, the PROC_ASLR_ACTIVE flag is or-ed with the value listed above.

PROC_PROTMAX_CTL Controls implicit application of PROT_MAX protection equal to the *prot* argument of the mmap(2) syscall, in the target process. The *data* parameter must point to the integer variable holding one of the following values:

PROC_PROTMAX_FORCE_ENABLE Enables implicit PROT_MAX application, even if it is disabled system-wide by the sysctl *vm.imply_prot_max*. The image flag might still prevent the enablement.

PROC_PROTMAX_FORCE_DISABLE Request that implicit application of PROT_MAX be disabled. Same notes as for PROC_PROTMAX_FORCE_ENABLE apply.

PROC_PROTMAX_NOFORCE Use the system-wide

configured policy for PROT_MAX.

PROC_PROTMAX_STATUS

Returns the current status of implicit PROT_MAX enablement for the target process. The *data* parameter must point to the integer variable, where one of the following values is written:

PROC_PROTMAX_FORCE_ENABLE

PROC_PROTMAX_FORCE_DISABLE

PROC_PROTMAX_NOFORCE

If the currently executed image in the process itself has implicit PROT_MAX application enabled, the PROC_PROTMAX_ACTIVE flag is or-ed with the value listed above.

PROC_SPROTECT Set process protection state. This is used to mark a process as protected from being killed if the system exhausts the available memory and swap. The *data* parameter must point to an integer containing an operation and zero or more optional flags. The following operations are supported:

PPROT_SET Mark the selected processes as protected.

PPROT_CLEAR Clear the protected state of selected processes.

The following optional flags are supported:

PPROT_DESCEND Apply the requested operation to all child processes of each selected process in addition to each selected process.

PPROT_INHERIT When used with PPROT_SET, mark all future child processes of each selected process as protected. Future child processes will also mark all of their future child processes.

PROC_REAP_ACQUIRE Acquires the reaper status for the current process. Reaper status means that children orphaned by the reaper’s descendants that were forked after

the acquisition of reaper status are reparented to the reaper process. After system initialization, init(8) is the default reaper.

PROC_REAP_RELEASE Release the reaper state for the current process. The reaper of the current process becomes the new reaper of the current process's descendants.

PROC_REAP_STATUS Provides information about the reaper of the specified process, or the process itself when it is a reaper. The *data* argument must point to a *procctl_reaper_status* structure which is filled in by the syscall on successful return.

```
struct procctl_reaper_status {
        u_int   rs_flags;
        u_int   rs_children;
        u_int   rs_descendants;
        pid_t   rs_reaper;
        pid_t   rs_pid;
};
```

The *rs_flags* may have the following flags returned:

REAPER_STATUS_OWNED The specified process has acquired reaper status and has not released it. When the flag is returned, the specified process *id*, pid, identifies the reaper, otherwise the *rs_reaper* field of the structure is set to the pid of the reaper for the specified process id.

REAPER_STATUS_REALINIT The specified process is the root of the reaper tree, i.e., init(8).

The *rs_children* field returns the number of children of the reaper among the descendants. It is possible to have a child whose reaper is not the specified process, since the reaper for any existing children is not reset on the PROC_REAP_ACQUIRE operation. The *rs_descendants* field returns the total number of descendants of the reaper(s), not counting descendants of the reaper in the subtree. The *rs_reaper* field returns the reaper pid. The *rs_pid* returns the pid of one reaper child if there are any descendants.

PROC_REAP_GETPIDS Queries the list of descendants of the reaper of the specified process. The request takes a pointer to a *procctl_reaper_pids* structure in the *data* parameter.

```
struct procctl_reaper_pids {
        u_int       rp_count;
        struct procctl_reaper_pidinfo *rp_pids;
};
```

When called, the *rp_pids* field must point to an array of *procctl_reaper_pidinfo* structures, to be filled in on return, and the *rp_count* field must specify the size of the array, into which no more than *rp_count* elements will be filled in by the kernel.

The *struct procctl_reaper_pidinfo* structure provides some information about one of the reaper's descendants. Note that for a descendant that is not a child, it may be incorrectly identified because of a race in which the original child process exited and the exited process's pid was reused for an unrelated process.

```
struct procctl_reaper_pidinfo {
        pid_t       pi_pid;
        pid_t       pi_subtree;
        u_int       pi_flags;
};
```

The *pi_pid* field is the process id of the descendant. The *pi_subtree* field provides the pid of the child of the reaper, which is the (grand-)parent of the process. The *pi_flags* field returns the following flags, further describing the descendant:

REAPER_PIDINFO_VALID Set to indicate that the *procctl_reaper_pidinfo* structure was filled in by the kernel. Zero-filling the *rp_pids* array and testing the REAPER_PIDINFO_VALID flag allows the caller to detect the end of the returned array.

REAPER_PIDINFO_CHILD The *pi_pid* field identifies the direct child of the reaper.

REAPER_PIDINFO_REAPER The reported process is itself a reaper. The descendants of the subordinate reaper are not reported.

REAPER_PIDINFO_ZOMBIE The reported process is in the zombie state, ready to be reaped.

REAPER_PIDINFO_STOPPED The reported process is stopped by a SIGSTOP/SIGTSTP signal.

REAPER_PIDINFO_EXITING The reported process is in the process of exiting (but not yet a zombie).

PROC_REAP_KILL Request to deliver a signal to some subset of the descendants of the reaper. The *data* parameter must point to a *procctl_reaper_kill* structure, which is used both for parameters and status return.

```
struct procctl_reaper_kill {
        int     rk_sig;
        u_int   rk_flags;
        pid_t   rk_subtree;
        u_int   rk_killed;
        pid_t   rk_fpid;
};
```

The *rk_sig* field specifies the signal to be delivered. Zero is not a valid signal number, unlike for kill(2). The *rk_flags* field further directs the operation. It is or-ed from the following flags:

REAPER_KILL_CHILDREN Deliver the specified signal only to direct children of the reaper.

REAPER_KILL_SUBTREE Deliver the specified signal only to descendants that were forked by the direct child with pid specified in the *rk_subtree* field.

If neither the REAPER_KILL_CHILDREN nor the REAPER_KILL_SUBTREE flags are specified, all current descendants of the reaper are signalled.

If a signal was delivered to any process, the return value from the request is zero. In this case, the *rk_killed* field identifies the number of processes signalled. The *rk_fpid* field is set to the pid of the first process for which signal delivery failed, e.g., due to permission problems. If no such process exists, the *rk_fpid* field is set to -1.

PROC_TRACE_CTL Enable or disable tracing of the specified process(es), according to the value of the integer argument. Tracing includes attachment to the process using the ptrace(2) and ktrace(2), debugging sysctls, hwpmc(4), dtrace(1), and core dumping. Possible values for the *data* argument are:

PROC_TRACE_CTL_ENABLE	Enable tracing, after it was disabled by PROC_TRACE_CTL_DISABLE. Only allowed for self.
PROC_TRACE_CTL_DISABLE	Disable tracing for the specified process. Tracing is re-enabled when the process changes the executing program with the execve(2) syscall. A child inherits the trace settings from the parent on fork(2).
PROC_TRACE_CTL_DISABLE_EXEC	Same as PROC_TRACE_CTL_DISABLE, but the setting persists for the process even after execve(2).

PROC_TRACE_STATUS Returns the current tracing status for the specified process in the integer variable pointed to by *data*. If tracing is disabled, *data* is set to -1. If tracing is enabled, but no debugger is attached by the ptrace(2) syscall, *data* is set to 0. If a debugger is attached, *data* is set to the pid of the debugger process.

PROC_TRAPCAP_CTL Controls the capability mode sandbox actions for the specified sandboxed processes, on a return from any syscall which gives either a ENOTCAPABLE or ECAPMODE error. If the control is enabled, such errors from the syscalls cause delivery of the synchronous SIGTRAP signal to the thread immediately before returning from the syscalls.

Possible values for the *data* argument are:

PROC_TRAPCAP_CTL_ENABLE
: Enable the SIGTRAP signal delivery on capability mode access violations. The enabled mode is inherited by the children of the process, and is kept after fexecve(2) calls.

PROC_TRAPCAP_CTL_DISABLE
: Disable the signal delivery on capability mode access violations. Note that the global sysctl kern.trap_enotcap might still cause the signal to be delivered. See capsicum(4).

On signal delivery, the *si_errno* member of the *siginfo* signal handler parameter is set to the syscall error value, and the *si_code* member is set to TRAP_CAP. The system call number is stored in the *si_syscall* field of the *siginfo* signal handler parameter. The other system call parameters can be read from the *ucontext_t* but the system call number is typically stored in the register that also contains the return value and so is unavailable in the signal handler.

See capsicum(4) for more information about the capability mode.

PROC_TRAPCAP_STATUS
: Return the current status of signalling capability mode access violations for the specified process. The integer value pointed to by the *data* argument is set to the PROC_TRAPCAP_CTL_ENABLE value if the process control enables signal delivery, and to PROC_TRAPCAP_CTL_DISABLE otherwise.

  See the note about sysctl kern.trap_enotcap above, which gives independent global control of signal delivery.

PROC_PDEATHSIG_CTL
: Request the delivery of a signal when the parent of the calling process exits. *idtype* must be P_PID and *id* must be the either caller's pid or zero, with no difference in effect. The value is cleared for child processes and when executing set-user-ID or set-group-ID binaries. *data* must point to a value of type *int* indicating the signal that should be delivered to the

caller. Use zero to cancel a previously requested signal delivery.

PROC_PDEATHSIG_STATUS

Query the current signal number that will be delivered when the parent of the calling process exits. *idtype* must be P_PID and *id* must be the either caller's pid or zero, with no difference in effect. *data* must point to a memory location that can hold a value of type *int*. If signal delivery has not been requested, it will contain zero on return.

PROC_STACKGAP_CTL

Controls the stack gaps in the specified process. A stack gap is the part of the growth area for a MAP_STACK mapped region that is reserved and never filled by memory. Instead, the process is guaranteed to receive a SIGSEGV signal on accessing pages in the gap. Gaps protect against stack overflow corrupting memory adjacent to the stack.

The *data* argument must point to an integer variable containing flags. The following flags are allowed:

PROC_STACKGAP_ENABLE

This flag is only accepted for consistency with PROC_STACKGAP_STATUS. If stack gaps are enabled, the flag is ignored. If disabled, the flag causes an EINVAL error to be returned. After gaps are disabled in a process, they can only be re-enabled when an execve(2) is performed.

PROC_STACKGAP_DISABLE

Disable stack gaps for the process. For existing stacks, the gap is no longer a reserved part of the growth area and can be filled by memory on access.

PROC_STACKGAP_ENABLE_EXEC

Enable stack gaps for programs started after an execve(2) by the specified process.

PROC_STACKGAP_DISABLE_EXEC

Inherit disabled stack gaps state

after execve(2). In other words, if the currently executing program has stack gaps disabled, they are kept disabled on exec. If gaps were enabled, they are kept enabled after exec.

The stack gap state is inherited from the parent on fork(2).

PROC_STACKGAP_STATUS

Returns the current stack gap state for the specified process. *data* must point to an integer variable, which is used to return a bitmask consisting of the following flags:

PROC_STACKGAP_ENABLE	Stack gaps are enabled.
PROC_STACKGAP_DISABLE	Stack gaps are disabled.
PROC_STACKGAP_ENABLE_EXEC	Stack gaps are enabled in the process after execve(2).
PROC_STACKGAP_DISABLE_EXEC	Stack gaps are disabled in the process after execve(2).

PROC_NO_NEW_PRIVS_CTL

Allows one to ignore the SUID and SGID bits on the program images activated by execve(2) in the specified process and its future descendants. The *data* parameter must point to the integer variable holding the following value:

PROC_NO_NEW_PRIVS_ENABLE	Request SUID and SGID bits to be ignored.

It is not possible to disable it once it has been enabled.

PROC_NO_NEW_PRIVS_STATUS

Returns the current status of SUID/SGID enablement for the target process. The *data* parameter must point to the integer variable, where one of the following values is written:

PROC_NO_NEW_PRIVS_ENABLE

PROC_NO_NEW_PRIVS_DISABLE

PROC_WXMAP_CTL Controls the ’write exclusive against execution’ permissions for the mappings in the process address space. It overrides the global settings established by the kern.elf{32/64}.allow_wx sysctl, and the corresponding bit in the ELF control note, see elfctl(1).

The *data* parameter must point to the integer variable holding one of the following values:

PROC_WX_MAPPINGS_PERMIT Enable creation of mappings that have both write and execute protection attributes, in the specified process’ address space.

PROC_WX_MAPPINGS_DISALLOW_EXEC In the new address space created by execve(2), disallow creation of mappings that have both write and execute permissions.

Once creation of writeable and executable mappings is allowed, it is impossible (and pointless) to disallow it. The only way to ensure the absence of such mappings after they were enabled in a given process, is to set the PROC_WX_MAPPINGS_DISALLOW_EXEC flag and execve(2) an image.

PROC_WXMAP_STATUS Returns the current status of the ’write exclusive against execution’ enforcement for the specified process. The data parameter must point to the integer variable, where one of the following values is written:

PROC_WX_MAPPINGS_PERMIT Creation of simultaneously writable and executable mapping is permitted, otherwise

the process cannot create such mappings.

PROC_WX_MAPPINGS_DISALLOW_EXEC After execve(2), the new address space should disallow creation of simultaneously writable and executable mappings.

Additionally, if the address space of the process disallows creation of simultaneously writable and executable mappings and it is guaranteed that no such mapping was created since address space creation, the PROC_WXORX_ENFORCE flag is set in the returned value.

x86 MACHINE-SPECIFIC REQUESTS

PROC_KPTI_CTL AMD64 only. Controls the Kernel Page Table Isolation (KPTI) option for the children of the specified process. For the command to work, the *vm.pmap.kpti* tunable must be enabled on boot. It is not possible to change the KPTI setting for a running process, except at the execve(2), where the address space is reinitialized.

The *data* parameter must point to an integer variable containing one of the following commands:

PROC_KPTI_CTL_ENABLE_ON_EXEC Enable KPTI after execve(2).

PROC_KPTI_CTL_DISABLE_ON_EXEC Disable KPTI after execve(2). Only root or a process having the *PRIV_IO* privilege might use this option.

PROC_KPTI_STATUS Returns the current KPTI status for the specified process. *data* must point to the integer variable, which returns the following statuses:

PROC_KPTI_CTL_ENABLE_ON_EXEC

PROC_KPTI_CTL_DISABLE_ON_EXEC

The status is or-ed with the *PROC_KPTI_STATUS_ACTIVE* in case KPTI is

active for the current address space of the process.

NOTES

Disabling tracing on a process should not be considered a security feature, as it is bypassable both by the kernel and privileged processes, and via other system mechanisms. As such, it should not be utilized to reliably protect cryptographic keying material or other confidential data.

Note that processes can trivially bypass the 'no simultaneously writable and executable mappings' policy by first marking some mapping as writeable and write code to it, then removing write and adding execute permission. This may be legitimately required by some programs, such as JIT compilers.

RETURN VALUES

If an error occurs, a value of -1 is returned and *errno* is set to indicate the error.

ERRORS

The **procctl**() system call will fail if:

[EFAULT] The *data* parameter points outside the process's allocated address space.

[EINVAL] The *cmd* argument specifies an unsupported command.

The *idtype* argument specifies an unsupported identifier type.

[EPERM] The calling process does not have permission to perform the requested operation on any of the selected processes.

[ESRCH] No processes matched the requested *idtype* and *id*.

[EINVAL] An invalid operation or flag was passed in *data* for a PROC_SPROTECT command.

[EPERM] The *idtype* argument is not equal to P_PID, or *id* is not equal to the pid of the calling process, for PROC_REAP_ACQUIRE or PROC_REAP_RELEASE requests.

[EINVAL] Invalid or undefined flags were passed to a PROC_REAP_KILL request.

[EINVAL] An invalid or zero signal number was requested for a PROC_REAP_KILL request.

[EINVAL] The PROC_REAP_RELEASE request was issued by the init(8) process.

[EBUSY] The PROC_REAP_ACQUIRE request was issued by a process that had already acquired reaper status and has not yet released it.

[EBUSY] The PROC_TRACE_CTL request was issued for a process already being traced.

[EPERM] The PROC_TRACE_CTL request to re-enable tracing of the process (PROC_TRACE_CTL_ENABLE), or to disable persistence of PROC_TRACE_CTL_DISABLE on execve(2) was issued for a non-current process.

[EINVAL] The value of the integer *data* parameter for the PROC_TRACE_CTL or PROC_TRAPCAP_CTL request is invalid.

[EINVAL] The PROC_PDEATHSIG_CTL or PROC_PDEATHSIG_STATUS request referenced an unsupported *id*, *idtype* or invalid signal number.

SEE ALSO

dtrace(1), proccontrol(1), protect(1), cap_enter(2), kill(2), ktrace(2), mmap(2), mprotect(2), ptrace(2), wait(2), capsicum(4), hwpmc(4), init(8)

HISTORY

The **procctl**() function appeared in FreeBSD 10.0.

The reaper facility is based on a similar feature of Linux and DragonflyBSD, and first appeared in FreeBSD 10.2.

The PROC_PDEATHSIG_CTL facility is based on the prctl(PR_SET_PDEATHSIG, ...) feature of Linux, and first appeared in FreeBSD 11.2.

The ASLR support was added to system for the checklists compliance in FreeBSD 13.0.