Rocky Enterprise Linux 9.2 Manual Pages on command 'ptrace.2'

C:\>man ptrace.2

PTRACE(2) Linux Programmer's Manual PTRACE(2)

NAME

ptrace - process trace

SYNOPSIS

#include <sys/ptrace.h>

long ptrace(enum __ptrace_request request, pid_t pid,

void *addr, void *data);

DESCRIPTION

The ptrace() system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.

A tracee first needs to be attached to the tracer. Attachment and subsequent com? mands are per thread: in a multithreaded process, every thread can be individually attached to a (potentially different) tracer, or left not attached and thus not de? bugged. Therefore, "tracee" always means "(one) thread", never "a (possibly multi? threaded) process". Ptrace commands are always sent to a specific tracee using a call of the form

ptrace(PTRACE_foo, pid, ...)

where pid is the thread ID of the corresponding Linux thread.

(Note that in this page, a "multithreaded process" means a thread group consisting of threads created using the clone(2) CLONE_THREAD flag.)

A process can initiate a trace by calling fork(2) and having the resulting child do a PTRACE_TRACEME, followed (typically) by an execve(2). Alternatively, one process may commence tracing another process using PTRACE_ATTACH or PTRACE_SEIZE.

While being traced, the tracee will stop each time a signal is delivered, even if the signal is being ignored. (An exception is SIGKILL, which has its usual effect.) The tracer will be notified at its next call to waitpid(2) (or one of the related "wait" system calls); that call will return a status value containing information that indicates the cause of the stop in the tracee. While the tracee is stopped, the tracer can use various ptrace requests to inspect and modify the tracee. The tracer then causes the tracee to continue, optionally ignoring the delivered signal (or even delivering a different signal instead).

If the PTRACE_O_TRACEEXEC option is not in effect, all successful calls to execve(2) by the traced process will cause it to be sent a SIGTRAP signal, giving the parent a chance to gain control before the new program begins execution.

When the tracer is finished tracing, it can cause the tracee to continue executing in a normal, untraced mode via PTRACE_DETACH.

The value of request determines the action to be performed:

PTRACE_TRACEME

Indicate that this process is to be traced by its parent. A process probably shouldn't make this request if its parent isn't expecting to trace it. (pid, addr, and data are ignored.)

The PTRACE_TRACEME request is used only by the tracee; the remaining requests are used only by the tracer. In the following requests, pid specifies the thread ID of the tracee to be acted on. For requests other than PTRACE_ATTACH, PTRACE_SEIZE, PTRACE_INTERRUPT, and PTRACE_KILL, the tracee must be stopped.

PTRACE_PEEKTEXT, PTRACE_PEEKDATA

Read a word at the address addr in the tracee's memory, returning the word as the result of the ptrace() call. Linux does not have separate text and data address spaces, so these two requests are currently equivalent. (data is ignored; but see NOTES.)

PTRACE_PEEKUSER

Read a word at offset addr in the tracee's USER area, which holds the regis?

ters and other information about the process (see <sys/user.h>). The word is returned as the result of the ptrace() call. Typically, the offset must be word-aligned, though this might vary by architecture. See NOTES. (data is ignored; but see NOTES.)

PTRACE_POKETEXT, PTRACE_POKEDATA

Copy the word data to the address addr in the tracee's memory. As for PTRACE_PEEKTEXT and PTRACE_PEEKDATA, these two requests are currently equiv? alent.

PTRACE_POKEUSER

Copy the word data to offset addr in the tracee's USER area. As for PTRACE_PEEKUSER, the offset must typically be word-aligned. In order to maintain the integrity of the kernel, some modifications to the USER area are disallowed.

PTRACE_GETREGS, PTRACE_GETFPREGS

Copy the tracee's general-purpose or floating-point registers, respectively, to the address data in the tracer. See <sys/user.h> for information on the format of this data. (addr is ignored.) Note that SPARC systems have the meaning of data and addr reversed; that is, data is ignored and the regis? ters are copied to the address addr. PTRACE_GETREGS and PTRACE_GETFPREGS are not present on all architectures.

PTRACE_GETREGSET (since Linux 2.6.34)

Read the tracee's registers. addr specifies, in an architecture-dependent way, the type of registers to be read. NT_PRSTATUS (with numerical value 1) usually results in reading of general-purpose registers. If the CPU has, for example, floating-point and/or vector registers, they can be retrieved by setting addr to the corresponding NT_foo constant. data points to a struct iovec, which describes the destination buffer's location and length. On return, the kernel modifies iov.len to indicate the actual number of bytes returned.

PTRACE_SETREGS, PTRACE_SETFPREGS

Modify the tracee's general-purpose or floating-point registers, respec? tively, from the address data in the tracer. As for PTRACE_POKEUSER, some general-purpose register modifications may be disallowed. (addr is ig?

nored.) Note that SPARC systems have the meaning of data and addr reversed;
that is, data is ignored and the registers are copied from the address addr.
PTRACE_SETREGS and PTRACE_SETFPREGS are not present on all architectures.

PTRACE_SETREGSET (since Linux 2.6.34)

Modify the tracee's registers. The meaning of addr and data is analogous to
PTRACE_GETREGSET.

PTRACE_GETSIGINFO (since Linux 2.3.99-pre6)

Retrieve information about the signal that caused the stop. Copy a sig?
info_t structure (see sigaction(2)) from the tracee to the address data in
the tracer. (addr is ignored.)

PTRACE_SETSIGINFO (since Linux 2.3.99-pre6)

Set signal information: copy a siginfo_t structure from the address data in
the tracer to the tracee. This will affect only signals that would normally
be delivered to the tracee and were caught by the tracer. It may be diffi?
cult to tell these normal signals from synthetic signals generated by
ptrace() itself. (addr is ignored.)

PTRACE_PEEKSIGINFO (since Linux 3.10)

Retrieve siginfo_t structures without removing signals from a queue. addr
points to a ptrace_peeksiginfo_args structure that specifies the ordinal po?
sition from which copying of signals should start, and the number of signals
to copy. siginfo_t structures are copied into the buffer pointed to by
data. The return value contains the number of copied signals (zero indi?
cates that there is no signal corresponding to the specified ordinal posi?
tion). Within the returned siginfo structures, the si_code field includes
information (__SI_CHLD, __SI_FAULT, etc.) that are not otherwise exposed to
user space.

```
struct ptrace_peeksiginfo_args {
    u64 off;    /* Ordinal position in queue at which
                   to start copying signals */
    u32 flags;  /* PTRACE_PEEKSIGINFO_SHARED or 0 */
    s32 nr;     /* Number of signals to copy */
};
```

Currently, there is only one flag, PTRACE_PEEKSIGINFO_SHARED, for dumping

signals from the process-wide signal queue. If this flag is not set, sig?
nals are read from the per-thread queue of the specified thread.

PTRACE_GETSIGMASK (since Linux 3.11)

Place a copy of the mask of blocked signals (see sigprocmask(2)) in the buf?
fer pointed to by data, which should be a pointer to a buffer of type
sigset_t. The addr argument contains the size of the buffer pointed to by
data (i.e., sizeof(sigset_t)).

PTRACE_SETSIGMASK (since Linux 3.11)

Change the mask of blocked signals (see sigprocmask(2)) to the value speci?
fied in the buffer pointed to by data, which should be a pointer to a buffer
of type sigset_t. The addr argument contains the size of the buffer pointed
to by data (i.e., sizeof(sigset_t)).

PTRACE_SETOPTIONS (since Linux 2.4.6; see BUGS for caveats)

Set ptrace options from data. (addr is ignored.) data is interpreted as a
bit mask of options, which are specified by the following flags:

PTRACE_O_EXITKILL (since Linux 3.8)

Send a SIGKILL signal to the tracee if the tracer exits. This option
is useful for ptrace jailers that want to ensure that tracees can
never escape the tracer's control.

PTRACE_O_TRACECLONE (since Linux 2.5.46)

Stop the tracee at the next clone(2) and automatically start tracing
the newly cloned process, which will start with a SIGSTOP, or
PTRACE_EVENT_STOP if PTRACE_SEIZE was used. A waitpid(2) by the
tracer will return a status value such that

status>>8 == (SIGTRAP | (PTRACE_EVENT_CLONE<<8))

The PID of the new process can be retrieved with PTRACE_GETEVENTMSG.
This option may not catch clone(2) calls in all cases. If the tracee
calls clone(2) with the CLONE_VFORK flag, PTRACE_EVENT_VFORK will be
delivered instead if PTRACE_O_TRACEVFORK is set; otherwise if the
tracee calls clone(2) with the exit signal set to SIGCHLD,
PTRACE_EVENT_FORK will be delivered if PTRACE_O_TRACEFORK is set.

PTRACE_O_TRACEEXEC (since Linux 2.5.46)

Stop the tracee at the next execve(2). A waitpid(2) by the tracer

will return a status value such that

status>>8 == (SIGTRAP | (PTRACE_EVENT_EXEC<<8))

If the execing thread is not a thread group leader, the thread ID is reset to thread group leader's ID before this stop. Since Linux 3.0, the former thread ID can be retrieved with PTRACE_GETEVENTMSG.

PTRACE_O_TRACEEXIT (since Linux 2.5.60)

Stop the tracee at exit. A waitpid(2) by the tracer will return a status value such that

status>>8 == (SIGTRAP | (PTRACE_EVENT_EXIT<<8))

The tracee's exit status can be retrieved with PTRACE_GETEVENTMSG.

The tracee is stopped early during process exit, when registers are still available, allowing the tracer to see where the exit occurred, whereas the normal exit notification is done after the process is finished exiting. Even though context is available, the tracer can? not prevent the exit from happening at this point.

PTRACE_O_TRACEFORK (since Linux 2.5.46)

Stop the tracee at the next fork(2) and automatically start tracing the newly forked process, which will start with a SIGSTOP, or PTRACE_EVENT_STOP if PTRACE_SEIZE was used. A waitpid(2) by the tracer will return a status value such that

status>>8 == (SIGTRAP | (PTRACE_EVENT_FORK<<8))

The PID of the new process can be retrieved with PTRACE_GETEVENTMSG.

PTRACE_O_TRACESYSGOOD (since Linux 2.4.6)

When delivering system call traps, set bit 7 in the signal number (i.e., deliver SIGTRAP|0x80). This makes it easy for the tracer to distinguish normal traps from those caused by a system call.

PTRACE_O_TRACEVFORK (since Linux 2.5.46)

Stop the tracee at the next vfork(2) and automatically start tracing the newly vforked process, which will start with a SIGSTOP, or PTRACE_EVENT_STOP if PTRACE_SEIZE was used. A waitpid(2) by the tracer will return a status value such that

status>>8 == (SIGTRAP | (PTRACE_EVENT_VFORK<<8))

The PID of the new process can be retrieved with PTRACE_GETEVENTMSG.

PTRACE_O_TRACEVFORKDONE (since Linux 2.5.60)

Stop the tracee at the completion of the next vfork(2). A waitpid(2) by the tracer will return a status value such that

status>>8 == (SIGTRAP | (PTRACE_EVENT_VFORK_DONE<<8))

The PID of the new process can (since Linux 2.6.18) be retrieved with PTRACE_GETEVENTMSG.

PTRACE_O_TRACESECCOMP (since Linux 3.5)

Stop the tracee when a seccomp(2) SECCOMP_RET_TRACE rule is triggered. A waitpid(2) by the tracer will return a status value such that

status>>8 == (SIGTRAP | (PTRACE_EVENT_SECCOMP<<8))

While this triggers a PTRACE_EVENT stop, it is similar to a syscall-enter-stop. For details, see the note on PTRACE_EVENT_SECCOMP below.

The seccomp event message data (from the SECCOMP_RET_DATA portion of the seccomp filter rule) can be retrieved with PTRACE_GETEVENTMSG.

PTRACE_O_SUSPEND_SECCOMP (since Linux 4.3)

Suspend the tracee's seccomp protections. This applies regardless of mode, and can be used when the tracee has not yet installed seccomp filters. That is, a valid use case is to suspend a tracee's seccomp protections before they are installed by the tracee, let the tracee install the filters, and then clear this flag when the filters should be resumed. Setting this option requires that the tracer have the CAP_SYS_ADMIN capability, not have any seccomp protections installed, and not have PTRACE_O_SUSPEND_SECCOMP set on itself.

PTRACE_GETEVENTMSG (since Linux 2.5.46)

Retrieve a message (as an unsigned long) about the ptrace event that just happened, placing it at the address data in the tracer. For PTRACE_EVENT_EXIT, this is the tracee's exit status. For PTRACE_EVENT_FORK, PTRACE_EVENT_VFORK, PTRACE_EVENT_VFORK_DONE, and PTRACE_EVENT_CLONE, this is the PID of the new process. For PTRACE_EVENT_SECCOMP, this is the seccomp(2) filter's SECCOMP_RET_DATA associated with the triggered rule. (addr is ignored.)

PTRACE_CONT

Restart the stopped tracee process. If data is nonzero, it is interpreted as the number of a signal to be delivered to the tracee; otherwise, no signal is delivered. Thus, for example, the tracer can control whether a signal sent to the tracee is delivered or not. (addr is ignored.)

PTRACE_SYSCALL, PTRACE_SINGLESTEP

Restart the stopped tracee as for PTRACE_CONT, but arrange for the tracee to be stopped at the next entry to or exit from a system call, or after execution of a single instruction, respectively. (The tracee will also, as usual, be stopped upon receipt of a signal.) From the tracer's perspective, the tracee will appear to have been stopped by receipt of a SIGTRAP. So, for PTRACE_SYSCALL, for example, the idea is to inspect the arguments to the system call at the first stop, then do another PTRACE_SYSCALL and inspect the return value of the system call at the second stop. The data argument is treated as for PTRACE_CONT. (addr is ignored.)

PTRACE_SYSEMU, PTRACE_SYSEMU_SINGLESTEP (since Linux 2.6.14)

For PTRACE_SYSEMU, continue and stop on entry to the next system call, which will not be executed. See the documentation on syscall-stops below. For PTRACE_SYSEMU_SINGLESTEP, do the same but also singlestep if not a system call. This call is used by programs like User Mode Linux that want to emulate all the tracee's system calls. The data argument is treated as for PTRACE_CONT. The addr argument is ignored. These requests are currently supported only on x86.

PTRACE_LISTEN (since Linux 3.4)

Restart the stopped tracee, but prevent it from executing. The resulting state of the tracee is similar to a process which has been stopped by a SIGSTOP (or other stopping signal). See the "group-stop" subsection for additional information. PTRACE_LISTEN works only on tracees attached by PTRACE_SEIZE.

PTRACE_KILL

Send the tracee a SIGKILL to terminate it. (addr and data are ignored.)

This operation is deprecated; do not use it! Instead, send a SIGKILL directly using kill(2) or tgkill(2). The problem with PTRACE_KILL is that it requires the tracee to be in signal-delivery-stop, otherwise it may not work

(i.e., may complete successfully but won't kill the tracee). By contrast, sending a SIGKILL directly has no such limitation.

PTRACE_INTERRUPT (since Linux 3.4)

Stop a tracee. If the tracee is running or sleeping in kernel space and PTRACE_SYSCALL is in effect, the system call is interrupted and syscall-exit-stop is reported. (The interrupted system call is restarted when the tracee is restarted.) If the tracee was already stopped by a signal and PTRACE_LISTEN was sent to it, the tracee stops with PTRACE_EVENT_STOP and WSTOPSIG(status) returns the stop signal. If any other ptrace-stop is gen? erated at the same time (for example, if a signal is sent to the tracee), this ptrace-stop happens. If none of the above applies (for example, if the tracee is running in user space), it stops with PTRACE_EVENT_STOP with WSTOPSIG(status) == SIGTRAP. PTRACE_INTERRUPT only works on tracees at? tached by PTRACE_SEIZE.

PTRACE_ATTACH

Attach to the process specified in pid, making it a tracee of the calling process. The tracee is sent a SIGSTOP, but will not necessarily have stopped by the completion of this call; use waitpid(2) to wait for the tracee to stop. See the "Attaching and detaching" subsection for additional information. (addr and data are ignored.)

Permission to perform a PTRACE_ATTACH is governed by a ptrace access mode PTRACE_MODE_ATTACH_REALCREDS check; see below.

PTRACE_SEIZE (since Linux 3.4)

Attach to the process specified in pid, making it a tracee of the calling process. Unlike PTRACE_ATTACH, PTRACE_SEIZE does not stop the process. Group-stops are reported as PTRACE_EVENT_STOP and WSTOPSIG(status) returns the stop signal. Automatically attached children stop with PTRACE_EVENT_STOP and WSTOPSIG(status) returns SIGTRAP instead of having SIGSTOP signal delivered to them. execve(2) does not deliver an extra SIG? TRAP. Only a PTRACE_SEIZEd process can accept PTRACE_INTERRUPT and PTRACE_LISTEN commands. The "seized" behavior just described is inherited by children that are automatically attached using PTRACE_O_TRACEFORK, PTRACE_O_TRACEVFORK, and PTRACE_O_TRACECLONE. addr must be zero. data con?

tains a bit mask of ptrace options to activate immediately.

Permission to perform a PTRACE_SEIZE is governed by a ptrace access mode PTRACE_MODE_ATTACH_REALCREDS check; see below.

PTRACE_SECCOMP_GET_FILTER (since Linux 4.4)

This operation allows the tracer to dump the tracee's classic BPF filters.

addr is an integer specifying the index of the filter to be dumped. The most recently installed filter has the index 0. If addr is greater than the number of installed filters, the operation fails with the error ENOENT.

data is either a pointer to a struct sock_filter array that is large enough to store the BPF program, or NULL if the program is not to be stored.

Upon success, the return value is the number of instructions in the BPF pro? gram. If data was NULL, then this return value can be used to correctly size the struct sock_filter array passed in a subsequent call.

This operation fails with the error EACCES if the caller does not have the CAP_SYS_ADMIN capability or if the caller is in strict or filter seccomp mode. If the filter referred to by addr is not a classic BPF filter, the operation fails with the error EMEDIUMTYPE.

This operation is available if the kernel was configured with both the CON? FIG_SECCOMP_FILTER and the CONFIG_CHECKPOINT_RESTORE options.

PTRACE_DETACH

Restart the stopped tracee as for PTRACE_CONT, but first detach from it. Under Linux, a tracee can be detached in this way regardless of which method was used to initiate tracing. (addr is ignored.)

PTRACE_GET_THREAD_AREA (since Linux 2.6.0)

This operation performs a similar task to get_thread_area(2). It reads the TLS entry in the GDT whose index is given in addr, placing a copy of the en? try into the struct user_desc pointed to by data. (By contrast with get_thread_area(2), the entry_number of the struct user_desc is ignored.)

PTRACE_SET_THREAD_AREA (since Linux 2.6.0)

This operation performs a similar task to set_thread_area(2). It sets the TLS entry in the GDT whose index is given in addr, assigning it the data supplied in the struct user_desc pointed to by data. (By contrast with set_thread_area(2), the entry_number of the struct user_desc is ignored; in

other words, this ptrace operation can't be used to allocate a free TLS en?
try.)

PTRACE_GET_SYSCALL_INFO (since Linux 5.3)

Retrieve information about the system call that caused the stop. The infor?
mation is placed into the buffer pointed by the data argument, which should
be a pointer to a buffer of type struct ptrace_syscall_info. The addr argu?
ment contains the size of the buffer pointed to by the data argument (i.e.,
sizeof(struct ptrace_syscall_info)). The return value contains the number
of bytes available to be written by the kernel. If the size of the data to
be written by the kernel exceeds the size specified by the addr argument,
the output data is truncated.

The ptrace_syscall_info structure contains the following fields:

```
struct ptrace_syscall_info {
    __u8 op;        /* Type of system call stop */
    __u32 arch;     /* AUDIT_ARCH_* value; see seccomp(2) */
    __u64 instruction_pointer; /* CPU instruction pointer */
    __u64 stack_pointer;       /* CPU stack pointer */
    union {
        struct {    /* op == PTRACE_SYSCALL_INFO_ENTRY */
            __u64 nr;          /* System call number */
            __u64 args[6];     /* System call arguments */
        } entry;
        struct {    /* op == PTRACE_SYSCALL_INFO_EXIT */
            __s64 rval;        /* System call return value */
            __u8 is_error;     /* System call error flag;
                                  Boolean: does rval contain
                                  an error value (-ERRCODE) or
                                  a nonerror return value? */
        } exit;
        struct {    /* op == PTRACE_SYSCALL_INFO_SECCOMP */
            __u64 nr;          /* System call number */
            __u64 args[6];     /* System call arguments */
            __u32 ret_data;    /* SECCOMP_RET_DATA portion
```

```
                        of SECCOMP_RET_TRACE
                        return value */
        } seccomp;
    };
};
```

The op, arch, instruction_pointer, and stack_pointer fields are defined for all kinds of ptrace system call stops. The rest of the structure is a union; one should read only those fields that are meaningful for the kind of system call stop specified by the op field.

The op field has one of the following values (defined in <linux/ptrace.h>) indicating what type of stop occurred and which part of the union is filled:

PTRACE_SYSCALL_INFO_ENTRY

The entry component of the union contains information relating to a system call entry stop.

PTRACE_SYSCALL_INFO_EXIT

The exit component of the union contains information relating to a system call exit stop.

PTRACE_SYSCALL_INFO_SECCOMP

The seccomp component of the union contains information relating to a PTRACE_EVENT_SECCOMP stop.

PTRACE_SYSCALL_INFO_NONE

No component of the union contains relevant information.

Death under ptrace

When a (possibly multithreaded) process receives a killing signal (one whose dispo? sition is set to SIG_DFL and whose default action is to kill the process), all threads exit. Tracees report their death to their tracer(s). Notification of this event is delivered via waitpid(2).

Note that the killing signal will first cause signal-delivery-stop (on one tracee only), and only after it is injected by the tracer (or after it was dispatched to a thread which isn't traced), will death from the signal happen on all tracees within a multithreaded process. (The term "signal-delivery-stop" is explained below.)

SIGKILL does not generate signal-delivery-stop and therefore the tracer can't sup? press it. SIGKILL kills even within system calls (syscall-exit-stop is not gener?

ated prior to death by SIGKILL). The net effect is that SIGKILL always kills the process (all its threads), even if some threads of the process are ptraced.

When the tracee calls _exit(2), it reports its death to its tracer. Other threads are not affected.

When any thread executes exit_group(2), every tracee in its thread group reports its death to its tracer.

If the PTRACE_O_TRACEEXIT option is on, PTRACE_EVENT_EXIT will happen before actual death. This applies to exits via exit(2), exit_group(2), and signal deaths (except SIGKILL, depending on the kernel version; see BUGS below), and when threads are torn down on execve(2) in a multithreaded process.

The tracer cannot assume that the ptrace-stopped tracee exists. There are many scenarios when the tracee may die while stopped (such as SIGKILL). Therefore, the tracer must be prepared to handle an ESRCH error on any ptrace operation. Unfortu? nately, the same error is returned if the tracee exists but is not ptrace-stopped (for commands which require a stopped tracee), or if it is not traced by the process which issued the ptrace call. The tracer needs to keep track of the stopped/running state of the tracee, and interpret ESRCH as "tracee died unexpect? edly" only if it knows that the tracee has been observed to enter ptrace-stop.

Note that there is no guarantee that waitpid(WNOHANG) will reliably report the tracee's death status if a ptrace operation returned ESRCH. waitpid(WNOHANG) may return 0 instead. In other words, the tracee may be "not yet fully dead", but al? ready refusing ptrace requests.

The tracer can't assume that the tracee always ends its life by reporting WIFEX? ITED(status) or WIFSIGNALED(status); there are cases where this does not occur. For example, if a thread other than thread group leader does an execve(2), it dis? appears; its PID will never be seen again, and any subsequent ptrace stops will be reported under the thread group leader's PID.

Stopped states

A tracee can be in two states: running or stopped. For the purposes of ptrace, a tracee which is blocked in a system call (such as read(2), pause(2), etc.) is nev? ertheless considered to be running, even if the tracee is blocked for a long time.

The state of the tracee after PTRACE_LISTEN is somewhat of a gray area: it is not in any ptrace-stop (ptrace commands won't work on it, and it will deliver wait?

pid(2) notifications), but it also may be considered "stopped" because it is not executing instructions (is not scheduled), and if it was in group-stop before PTRACE_LISTEN, it will not respond to signals until SIGCONT is received.

There are many kinds of states when the tracee is stopped, and in ptrace discussions they are often conflated. Therefore, it is important to use precise terms.

In this manual page, any stopped state in which the tracee is ready to accept ptrace commands from the tracer is called ptrace-stop. Ptrace-stops can be further subdivided into signal-delivery-stop, group-stop, syscall-stop, PTRACE_EVENT stops, and so on. These stopped states are described in detail below.

When the running tracee enters ptrace-stop, it notifies its tracer using waitpid(2) (or one of the other "wait" system calls). Most of this manual page assumes that the tracer waits with:

```
pid = waitpid(pid_or_minus_1, &status, __WALL);
```

Ptrace-stopped tracees are reported as returns with pid greater than 0 and WIFSTOPPED(status) true.

The __WALL flag does not include the WSTOPPED and WEXITED flags, but implies their functionality.

Setting the WCONTINUED flag when calling waitpid(2) is not recommended: the "continued" state is per-process and consuming it can confuse the real parent of the tracee.

Use of the WNOHANG flag may cause waitpid(2) to return 0 ("no wait results available yet") even if the tracer knows there should be a notification. Example:

```
errno = 0;
ptrace(PTRACE_CONT, pid, 0L, 0L);
if (errno == ESRCH) {
    /* tracee is dead */
    r = waitpid(tracee, &status, __WALL | WNOHANG);
    /* r can still be 0 here! */
}
```

The following kinds of ptrace-stops exist: signal-delivery-stops, group-stops, PTRACE_EVENT stops, syscall-stops. They all are reported by waitpid(2) with WIFSTOPPED(status) true. They may be differentiated by examining the value status>>8, and if there is ambiguity in that value, by querying PTRACE_GETSIGINFO. (Note: the

WSTOPSIG(status) macro can't be used to perform this examination, because it re?
turns the value (status>>8) & 0xff.)

Signal-delivery-stop

When a (possibly multithreaded) process receives any signal except SIGKILL, the kernel selects an arbitrary thread which handles the signal. (If the signal is generated with tgkill(2), the target thread can be explicitly selected by the caller.) If the selected thread is traced, it enters signal-delivery-stop. At this point, the signal is not yet delivered to the process, and can be suppressed by the tracer. If the tracer doesn't suppress the signal, it passes the signal to the tracee in the next ptrace restart request. This second step of signal delivery is called signal injection in this manual page. Note that if the signal is blocked, signal-delivery-stop doesn't happen until the signal is unblocked, with the usual exception that SIGSTOP can't be blocked.

Signal-delivery-stop is observed by the tracer as waitpid(2) returning with WIF? STOPPED(status) true, with the signal returned by WSTOPSIG(status). If the signal is SIGTRAP, this may be a different kind of ptrace-stop; see the "Syscall-stops" and "execve" sections below for details. If WSTOPSIG(status) returns a stopping signal, this may be a group-stop; see below.

Signal injection and suppression

After signal-delivery-stop is observed by the tracer, the tracer should restart the tracee with the call

```
ptrace(PTRACE_restart, pid, 0, sig)
```

where PTRACE_restart is one of the restarting ptrace requests. If sig is 0, then a signal is not delivered. Otherwise, the signal sig is delivered. This operation is called signal injection in this manual page, to distinguish it from signal-de? livery-stop.

The sig value may be different from the WSTOPSIG(status) value: the tracer can cause a different signal to be injected.

Note that a suppressed signal still causes system calls to return prematurely. In this case, system calls will be restarted: the tracer will observe the tracee to reexecute the interrupted system call (or restart_syscall(2) system call for a few system calls which use a different mechanism for restarting) if the tracer uses PTRACE_SYSCALL. Even system calls (such as poll(2)) which are not restartable af?

ter signal are restarted after signal is suppressed; however, kernel bugs exist which cause some system calls to fail with EINTR even though no observable signal is injected to the tracee.

Restarting ptrace commands issued in ptrace-stops other than signal-delivery-stop are not guaranteed to inject a signal, even if sig is nonzero. No error is re? ported; a nonzero sig may simply be ignored. Ptrace users should not try to "cre? ate a new signal" this way: use tgkill(2) instead.

The fact that signal injection requests may be ignored when restarting the tracee after ptrace stops that are not signal-delivery-stops is a cause of confusion among ptrace users. One typical scenario is that the tracer observes group-stop, mis? takes it for signal-delivery-stop, restarts the tracee with

ptrace(PTRACE_restart, pid, 0, stopsig)

with the intention of injecting stopsig, but stopsig gets ignored and the tracee continues to run.

The SIGCONT signal has a side effect of waking up (all threads of) a group-stopped process. This side effect happens before signal-delivery-stop. The tracer can't suppress this side effect (it can only suppress signal injection, which only causes the SIGCONT handler to not be executed in the tracee, if such a handler is in? stalled). In fact, waking up from group-stop may be followed by signal-delivery- stop for signal(s) other than SIGCONT, if they were pending when SIGCONT was deliv? ered. In other words, SIGCONT may be not the first signal observed by the tracee after it was sent.

Stopping signals cause (all threads of) a process to enter group-stop. This side effect happens after signal injection, and therefore can be suppressed by the tracer.

In Linux 2.4 and earlier, the SIGSTOP signal can't be injected.

PTRACE_GETSIGINFO can be used to retrieve a siginfo_t structure which corresponds to the delivered signal. PTRACE_SETSIGINFO may be used to modify it. If PTRACE_SETSIGINFO has been used to alter siginfo_t, the si_signo field and the sig parameter in the restarting command must match, otherwise the result is undefined.

Group-stop

When a (possibly multithreaded) process receives a stopping signal, all threads stop. If some threads are traced, they enter a group-stop. Note that the stopping

signal will first cause signal-delivery-stop (on one tracee only), and only after it is injected by the tracer (or after it was dispatched to a thread which isn't traced), will group-stop be initiated on all tracees within the multithreaded process. As usual, every tracee reports its group-stop separately to the corre? sponding tracer.

Group-stop is observed by the tracer as waitpid(2) returning with WIFSTOPPED(sta? tus) true, with the stopping signal available via WSTOPSIG(status). The same re? sult is returned by some other classes of ptrace-stops, therefore the recommended practice is to perform the call

ptrace(PTRACE_GETSIGINFO, pid, 0, &siginfo)

The call can be avoided if the signal is not SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU; only these four signals are stopping signals. If the tracer sees something else, it can't be a group-stop. Otherwise, the tracer needs to call PTRACE_GETSIGINFO.

If PTRACE_GETSIGINFO fails with EINVAL, then it is definitely a group-stop. (Other failure codes are possible, such as ESRCH ("no such process") if a SIGKILL killed the tracee.)

If tracee was attached using PTRACE_SEIZE, group-stop is indicated by PTRACE_EVENT_STOP: status>>16 == PTRACE_EVENT_STOP. This allows detection of group-stops without requiring an extra PTRACE_GETSIGINFO call.

As of Linux 2.6.38, after the tracer sees the tracee ptrace-stop and until it restarts or kills it, the tracee will not run, and will not send notifications (ex? cept SIGKILL death) to the tracer, even if the tracer enters into another wait? pid(2) call.

The kernel behavior described in the previous paragraph causes a problem with transparent handling of stopping signals. If the tracer restarts the tracee after group-stop, the stopping signal is effectively ignored?the tracee doesn't remain stopped, it runs. If the tracer doesn't restart the tracee before entering into the next waitpid(2), future SIGCONT signals will not be reported to the tracer; this would cause the SIGCONT signals to have no effect on the tracee.

Since Linux 3.4, there is a method to overcome this problem: instead of PTRACE_CONT, a PTRACE_LISTEN command can be used to restart a tracee in a way where it does not execute, but waits for a new event which it can report via waitpid(2) (such as when it is restarted by a SIGCONT).

PTRACE_EVENT stops

If the tracer sets PTRACE_O_TRACE_* options, the tracee will enter ptrace-stops called PTRACE_EVENT stops.

PTRACE_EVENT stops are observed by the tracer as waitpid(2) returning with WIF? STOPPED(status), and WSTOPSIG(status) returns SIGTRAP (or for PTRACE_EVENT_STOP, returns the stopping signal if tracee is in a group-stop). An additional bit is set in the higher byte of the status word: the value status>>8 will be

((PTRACE_EVENT_foo<<8) | SIGTRAP).

The following events exist:

PTRACE_EVENT_VFORK

Stop before return from vfork(2) or clone(2) with the CLONE_VFORK flag. When the tracee is continued after this stop, it will wait for child to exit/exec before continuing its execution (in other words, the usual behav? ior on vfork(2)).

PTRACE_EVENT_FORK

Stop before return from fork(2) or clone(2) with the exit signal set to SIGCHLD.

PTRACE_EVENT_CLONE

Stop before return from clone(2).

PTRACE_EVENT_VFORK_DONE

Stop before return from vfork(2) or clone(2) with the CLONE_VFORK flag, but after the child unblocked this tracee by exiting or execing.

For all four stops described above, the stop occurs in the parent (i.e., the tracee), not in the newly created thread. PTRACE_GETEVENTMSG can be used to re? trieve the new thread's ID.

PTRACE_EVENT_EXEC

Stop before return from execve(2). Since Linux 3.0, PTRACE_GETEVENTMSG re? turns the former thread ID.

PTRACE_EVENT_EXIT

Stop before exit (including death from exit_group(2)), signal death, or exit caused by execve(2) in a multithreaded process. PTRACE_GETEVENTMSG returns the exit status. Registers can be examined (unlike when "real" exit hap? pens). The tracee is still alive; it needs to be PTRACE_CONTed or

PTRACE_DETACHed to finish exiting.

PTRACE_EVENT_STOP

Stop induced by PTRACE_INTERRUPT command, or group-stop, or initial ptrace-stop when a new child is attached (only if attached using PTRACE_SEIZE).

PTRACE_EVENT_SECCOMP

Stop triggered by a seccomp(2) rule on tracee syscall entry when PTRACE_O_TRACESECCOMP has been set by the tracer. The seccomp event message data (from the SECCOMP_RET_DATA portion of the seccomp filter rule) can be retrieved with PTRACE_GETEVENTMSG. The semantics of this stop are described in detail in a separate section below.

PTRACE_GETSIGINFO on PTRACE_EVENT stops returns SIGTRAP in si_signo, with si_code set to (event<<8) | SIGTRAP.

Syscall-stops

If the tracee was restarted by PTRACE_SYSCALL or PTRACE_SYSEMU, the tracee enters syscall-enter-stop just prior to entering any system call (which will not be exe? cuted if the restart was using PTRACE_SYSEMU, regardless of any change made to reg? isters at this point or how the tracee is restarted after this stop). No matter which method caused the syscall-entry-stop, if the tracer restarts the tracee with PTRACE_SYSCALL, the tracee enters syscall-exit-stop when the system call is fin? ished, or if it is interrupted by a signal. (That is, signal-delivery-stop never happens between syscall-enter-stop and syscall-exit-stop; it happens after syscall-exit-stop.). If the tracee is continued using any other method (including PTRACE_SYSEMU), no syscall-exit-stop occurs. Note that all mentions PTRACE_SYSEMU apply equally to PTRACE_SYSEMU_SINGLESTEP.

However, even if the tracee was continued using PTRACE_SYSCALL, it is not guaran? teed that the next stop will be a syscall-exit-stop. Other possibilities are that the tracee may stop in a PTRACE_EVENT stop (including seccomp stops), exit (if it entered _exit(2) or exit_group(2)), be killed by SIGKILL, or die silently (if it is a thread group leader, the execve(2) happened in another thread, and that thread is not traced by the same tracer; this situation is discussed later).

Syscall-enter-stop and syscall-exit-stop are observed by the tracer as waitpid(2) returning with WIFSTOPPED(status) true, and WSTOPSIG(status) giving SIGTRAP. If the PTRACE_O_TRACESYSGOOD option was set by the tracer, then WSTOPSIG(status) will

give the value (SIGTRAP | 0x80).

Syscall-stops can be distinguished from signal-delivery-stop with SIGTRAP by query?
ing PTRACE_GETSIGINFO for the following cases:

si_code <= 0

SIGTRAP was delivered as a result of a user-space action, for example, a system call (tgkill(2), kill(2), sigqueue(3), etc.), expiration of a POSIX timer, change of state on a POSIX message queue, or completion of an asyn? chronous I/O request.

si_code == SI_KERNEL (0x80)

SIGTRAP was sent by the kernel.

si_code == SIGTRAP or si_code == (SIGTRAP|0x80)

This is a syscall-stop.

However, syscall-stops happen very often (twice per system call), and performing PTRACE_GETSIGINFO for every syscall-stop may be somewhat expensive.

Some architectures allow the cases to be distinguished by examining registers. For example, on x86, rax == -ENOSYS in syscall-enter-stop. Since SIGTRAP (like any other signal) always happens after syscall-exit-stop, and at this point rax almost never contains -ENOSYS, the SIGTRAP looks like "syscall-stop which is not syscall-enter-stop"; in other words, it looks like a "stray syscall-exit-stop" and can be detected this way. But such detection is fragile and is best avoided.

Using the PTRACE_O_TRACESYSGOOD option is the recommended method to distinguish syscall-stops from other kinds of ptrace-stops, since it is reliable and does not incur a performance penalty.

Syscall-enter-stop and syscall-exit-stop are indistinguishable from each other by the tracer. The tracer needs to keep track of the sequence of ptrace-stops in or? der to not misinterpret syscall-enter-stop as syscall-exit-stop or vice versa. In general, a syscall-enter-stop is always followed by syscall-exit-stop, PTRACE_EVENT stop, or the tracee's death; no other kinds of ptrace-stop can occur in between.

However, note that seccomp stops (see below) can cause syscall-exit-stops, without preceding syscall-entry-stops. If seccomp is in use, care needs to be taken not to misinterpret such stops as syscall-entry-stops.

If after syscall-enter-stop, the tracer uses a restarting command other than PTRACE_SYSCALL, syscall-exit-stop is not generated.

PTRACE_GETSIGINFO on syscall-stops returns SIGTRAP in si_signo, with si_code set to SIGTRAP or (SIGTRAP|0x80).

PTRACE_EVENT_SECCOMP stops (Linux 3.5 to 4.7)

The behavior of PTRACE_EVENT_SECCOMP stops and their interaction with other kinds of ptrace stops has changed between kernel versions. This documents the behavior from their introduction until Linux 4.7 (inclusive). The behavior in later kernel versions is documented in the next section.

A PTRACE_EVENT_SECCOMP stop occurs whenever a SECCOMP_RET_TRACE rule is triggered. This is independent of which methods was used to restart the system call. Notably, seccomp still runs even if the tracee was restarted using PTRACE_SYSEMU and this system call is unconditionally skipped.

Restarts from this stop will behave as if the stop had occurred right before the system call in question. In particular, both PTRACE_SYSCALL and PTRACE_SYSEMU will normally cause a subsequent syscall-entry-stop. However, if after the PTRACE_EVENT_SECCOMP the system call number is negative, both the syscall-entry-stop and the system call itself will be skipped. This means that if the system call number is negative after a PTRACE_EVENT_SECCOMP and the tracee is restarted using PTRACE_SYSCALL, the next observed stop will be a syscall-exit-stop, rather than the syscall-entry-stop that might have been expected.

PTRACE_EVENT_SECCOMP stops (since Linux 4.8)

Starting with Linux 4.8, the PTRACE_EVENT_SECCOMP stop was reordered to occur be? tween syscall-entry-stop and syscall-exit-stop. Note that seccomp no longer runs (and no PTRACE_EVENT_SECCOMP will be reported) if the system call is skipped due to PTRACE_SYSEMU.

Functionally, a PTRACE_EVENT_SECCOMP stop functions comparably to a syscall-entry-stop (i.e., continuations using PTRACE_SYSCALL will cause syscall-exit-stops, the system call number may be changed and any other modified registers are visible to the to-be-executed system call as well). Note that there may be, but need not have been a preceding syscall-entry-stop.

After a PTRACE_EVENT_SECCOMP stop, seccomp will be rerun, with a SECCOMP_RET_TRACE rule now functioning the same as a SECCOMP_RET_ALLOW. Specifically, this means that if registers are not modified during the PTRACE_EVENT_SECCOMP stop, the system call will then be allowed.

PTRACE_SINGLESTEP stops

[Details of these kinds of stops are yet to be documented.]

Informational and restarting ptrace commands

Most ptrace commands (all except PTRACE_ATTACH, PTRACE_SEIZE, PTRACE_TRACEME, PTRACE_INTERRUPT, and PTRACE_KILL) require the tracee to be in a ptrace-stop, oth? erwise they fail with ESRCH.

When the tracee is in ptrace-stop, the tracer can read and write data to the tracee using informational commands. These commands leave the tracee in ptrace-stopped state:

```
ptrace(PTRACE_PEEKTEXT/PEEKDATA/PEEKUSER, pid, addr, 0);
ptrace(PTRACE_POKETEXT/POKEDATA/POKEUSER, pid, addr, long_val);
ptrace(PTRACE_GETREGS/GETFPREGS, pid, 0, &struct);
ptrace(PTRACE_SETREGS/SETFPREGS, pid, 0, &struct);
ptrace(PTRACE_GETREGSET, pid, NT_foo, &iov);
ptrace(PTRACE_SETREGSET, pid, NT_foo, &iov);
ptrace(PTRACE_GETSIGINFO, pid, 0, &siginfo);
ptrace(PTRACE_SETSIGINFO, pid, 0, &siginfo);
ptrace(PTRACE_GETEVENTMSG, pid, 0, &long_var);
ptrace(PTRACE_SETOPTIONS, pid, 0, PTRACE_O_flags);
```

Note that some errors are not reported. For example, setting signal information (siginfo) may have no effect in some ptrace-stops, yet the call may succeed (return 0 and not set errno); querying PTRACE_GETEVENTMSG may succeed and return some ran? dom value if current ptrace-stop is not documented as returning a meaningful event message.

The call

```
ptrace(PTRACE_SETOPTIONS, pid, 0, PTRACE_O_flags);
```

affects one tracee. The tracee's current flags are replaced. Flags are inherited by new tracees created and "auto-attached" via active PTRACE_O_TRACEFORK, PTRACE_O_TRACEVFORK, or PTRACE_O_TRACECLONE options.

Another group of commands makes the ptrace-stopped tracee run. They have the form:

```
ptrace(cmd, pid, 0, sig);
```

where cmd is PTRACE_CONT, PTRACE_LISTEN, PTRACE_DETACH, PTRACE_SYSCALL, PTRACE_SIN? GLESTEP, PTRACE_SYSEMU, or PTRACE_SYSEMU_SINGLESTEP. If the tracee is in signal-

delivery-stop, sig is the signal to be injected (if it is nonzero). Otherwise, sig may be ignored. (When restarting a tracee from a ptrace-stop other than signal-de? livery-stop, recommended practice is to always pass 0 in sig.)

Attaching and detaching

A thread can be attached to the tracer using the call

```
ptrace(PTRACE_ATTACH, pid, 0, 0);
```

or

```
ptrace(PTRACE_SEIZE, pid, 0, PTRACE_O_flags);
```

PTRACE_ATTACH sends SIGSTOP to this thread. If the tracer wants this SIGSTOP to have no effect, it needs to suppress it. Note that if other signals are concur? rently sent to this thread during attach, the tracer may see the tracee enter sig? nal-delivery-stop with other signal(s) first! The usual practice is to reinject these signals until SIGSTOP is seen, then suppress SIGSTOP injection. The design bug here is that a ptrace attach and a concurrently delivered SIGSTOP may race and the concurrent SIGSTOP may be lost.

Since attaching sends SIGSTOP and the tracer usually suppresses it, this may cause a stray EINTR return from the currently executing system call in the tracee, as de? scribed in the "Signal injection and suppression" section.

Since Linux 3.4, PTRACE_SEIZE can be used instead of PTRACE_ATTACH. PTRACE_SEIZE does not stop the attached process. If you need to stop it after attach (or at any other time) without sending it any signals, use PTRACE_INTERRUPT command.

The request

```
ptrace(PTRACE_TRACEME, 0, 0, 0);
```

turns the calling thread into a tracee. The thread continues to run (doesn't enter ptrace-stop). A common practice is to follow the PTRACE_TRACEME with

```
raise(SIGSTOP);
```

and allow the parent (which is our tracer now) to observe our signal-delivery-stop.

If the PTRACE_O_TRACEFORK, PTRACE_O_TRACEVFORK, or PTRACE_O_TRACECLONE options are in effect, then children created by, respectively, vfork(2) or clone(2) with the CLONE_VFORK flag, fork(2) or clone(2) with the exit signal set to SIGCHLD, and other kinds of clone(2), are automatically attached to the same tracer which traced their parent. SIGSTOP is delivered to the children, causing them to enter signal-delivery-stop after they exit the system call which created them.

Detaching of the tracee is performed by:

ptrace(PTRACE_DETACH, pid, 0, sig);

PTRACE_DETACH is a restarting operation; therefore it requires the tracee to be in ptrace-stop. If the tracee is in signal-delivery-stop, a signal can be injected. Otherwise, the sig parameter may be silently ignored.

If the tracee is running when the tracer wants to detach it, the usual solution is to send SIGSTOP (using tgkill(2), to make sure it goes to the correct thread), wait for the tracee to stop in signal-delivery-stop for SIGSTOP and then detach it (sup? pressing SIGSTOP injection). A design bug is that this can race with concurrent SIGSTOPs. Another complication is that the tracee may enter other ptrace-stops and needs to be restarted and waited for again, until SIGSTOP is seen. Yet another complication is to be sure that the tracee is not already ptrace-stopped, because no signal delivery happens while it is?not even SIGSTOP.

If the tracer dies, all tracees are automatically detached and restarted, unless they were in group-stop. Handling of restart from group-stop is currently buggy, but the "as planned" behavior is to leave tracee stopped and waiting for SIGCONT.

If the tracee is restarted from signal-delivery-stop, the pending signal is in? jected.

execve(2) under ptrace

When one thread in a multithreaded process calls execve(2), the kernel destroys all other threads in the process, and resets the thread ID of the execing thread to the thread group ID (process ID). (Or, to put things another way, when a multithreaded process does an execve(2), at completion of the call, it appears as though the ex? ecve(2) occurred in the thread group leader, regardless of which thread did the ex? ecve(2).) This resetting of the thread ID looks very confusing to tracers:

* All other threads stop in PTRACE_EVENT_EXIT stop, if the PTRACE_O_TRACEEXIT op? tion was turned on. Then all other threads except the thread group leader re? port death as if they exited via _exit(2) with exit code 0.
* The execing tracee changes its thread ID while it is in the execve(2). (Remem? ber, under ptrace, the "pid" returned from waitpid(2), or fed into ptrace calls, is the tracee's thread ID.) That is, the tracee's thread ID is reset to be the same as its process ID, which is the same as the thread group leader's thread ID.

* Then a PTRACE_EVENT_EXEC stop happens, if the PTRACE_O_TRACEEXEC option was turned on.

* If the thread group leader has reported its PTRACE_EVENT_EXIT stop by this time, it appears to the tracer that the dead thread leader "reappears from nowhere". (Note: the thread group leader does not report death via WIFEXITED(status) until there is at least one other live thread. This eliminates the possibility that the tracer will see it dying and then reappearing.) If the thread group leader was still alive, for the tracer this may look as if thread group leader returns from a different system call than it entered, or even "returned from a system call even though it was not in any system call". If the thread group leader was not traced (or was traced by a different tracer), then during execve(2) it will appear as if it has become a tracee of the tracer of the execing tracee.

All of the above effects are the artifacts of the thread ID change in the tracee.

The PTRACE_O_TRACEEXEC option is the recommended tool for dealing with this situa?tion. First, it enables PTRACE_EVENT_EXEC stop, which occurs before execve(2) re?turns. In this stop, the tracer can use PTRACE_GETEVENTMSG to retrieve the tracee's former thread ID. (This feature was introduced in Linux 3.0.) Second, the PTRACE_O_TRACEEXEC option disables legacy SIGTRAP generation on execve(2).

When the tracer receives PTRACE_EVENT_EXEC stop notification, it is guaranteed that except this tracee and the thread group leader, no other threads from the process are alive.

On receiving the PTRACE_EVENT_EXEC stop notification, the tracer should clean up all its internal data structures describing the threads of this process, and retain only one data structure?one which describes the single still running tracee, with thread ID == thread group ID == process ID.

Example: two threads call execve(2) at the same time:

*** we get syscall-enter-stop in thread 1: **

PID1 execve("/bin/foo", "foo" <unfinished ...>

*** we issue PTRACE_SYSCALL for thread 1 **

*** we get syscall-enter-stop in thread 2: **

PID2 execve("/bin/bar", "bar" <unfinished ...>

*** we issue PTRACE_SYSCALL for thread 2 **

*** we get PTRACE_EVENT_EXEC for PID0, we issue PTRACE_SYSCALL **

*** we get syscall-exit-stop for PID0: **

PID0 <... execve resumed>) = 0

If the PTRACE_O_TRACEEXEC option is not in effect for the execing tracee, and if the tracee was PTRACE_ATTACHed rather that PTRACE_SEIZEd, the kernel delivers an extra SIGTRAP to the tracee after execve(2) returns. This is an ordinary signal (similar to one which can be generated by kill -TRAP), not a special kind of ptrace-stop. Employing PTRACE_GETSIGINFO for this signal returns si_code set to 0 (SI_USER). This signal may be blocked by signal mask, and thus may be delivered (much) later.

Usually, the tracer (for example, strace(1)) would not want to show this extra post-execve SIGTRAP signal to the user, and would suppress its delivery to the tracee (if SIGTRAP is set to SIG_DFL, it is a killing signal). However, determin? ing which SIGTRAP to suppress is not easy. Setting the PTRACE_O_TRACEEXEC option or using PTRACE_SEIZE and thus suppressing this extra SIGTRAP is the recommended approach.

Real parent

The ptrace API (ab)uses the standard UNIX parent/child signaling over waitpid(2). This used to cause the real parent of the process to stop receiving several kinds of waitpid(2) notifications when the child process is traced by some other process. Many of these bugs have been fixed, but as of Linux 2.6.38 several still exist; see BUGS below.

As of Linux 2.6.38, the following is believed to work correctly:

* exit/death by signal is reported first to the tracer, then, when the tracer con? sumes the waitpid(2) result, to the real parent (to the real parent only when the whole multithreaded process exits). If the tracer and the real parent are the same process, the report is sent only once.

RETURN VALUE

On success, the PTRACE_PEEK* requests return the requested data (but see NOTES), the PTRACE_SECCOMP_GET_FILTER request returns the number of instructions in the BPF program, and other requests return zero.

On error, all requests return -1, and errno is set appropriately. Since the value returned by a successful PTRACE_PEEK* request may be -1, the caller must clear er? rno before the call, and then check it afterward to determine whether or not an er?

ror occurred.

ERRORS

EBUSY (i386 only) There was an error with allocating or freeing a debug register.

EFAULT There was an attempt to read from or write to an invalid area in the tracer's or the tracee's memory, probably because the area wasn't mapped or accessible. Unfortunately, under Linux, different variations of this fault will return EIO or EFAULT more or less arbitrarily.

EINVAL An attempt was made to set an invalid option.

EIO request is invalid, or an attempt was made to read from or write to an in? valid area in the tracer's or the tracee's memory, or there was a word-alignment violation, or an invalid signal was specified during a restart re? quest.

EPERM The specified process cannot be traced. This could be because the tracer has insufficient privileges (the required capability is CAP_SYS_PTRACE); un? privileged processes cannot trace processes that they cannot send signals to or those running set-user-ID/set-group-ID programs, for obvious reasons. Alternatively, the process may already be being traced, or (on kernels be? fore 2.6.26) be init(1) (PID 1).

ESRCH The specified process does not exist, or is not currently being traced by the caller, or is not stopped (for requests that require a stopped tracee).

CONFORMING TO

SVr4, 4.3BSD.

NOTES

Although arguments to ptrace() are interpreted according to the prototype given, glibc currently declares ptrace() as a variadic function with only the request ar? gument fixed. It is recommended to always supply four arguments, even if the re? quested operation does not use them, setting unused/ignored arguments to 0L or (void *) 0.

In Linux kernels before 2.6.26, init(1), the process with PID 1, may not be traced.

A tracees parent continues to be the tracer even if that tracer calls execve(2).

The layout of the contents of memory and the USER area are quite operating-system- and architecture-specific. The offset supplied, and the data returned, might not entirely match with the definition of struct user.

The size of a "word" is determined by the operating-system variant (e.g., for 32-bit Linux it is 32 bits).

This page documents the way the ptrace() call works currently in Linux. Its behav? ior differs significantly on other flavors of UNIX. In any case, use of ptrace() is highly specific to the operating system and architecture.

Ptrace access mode checking

Various parts of the kernel-user-space API (not just ptrace() operations), require so-called "ptrace access mode" checks, whose outcome determines whether an opera? tion is permitted (or, in a few cases, causes a "read" operation to return sani? tized data). These checks are performed in cases where one process can inspect sensitive information about, or in some cases modify the state of, another process.

The checks are based on factors such as the credentials and capabilities of the two processes, whether or not the "target" process is dumpable, and the results of checks performed by any enabled Linux Security Module (LSM)?for example, SELinux, Yama, or Smack?and by the commoncap LSM (which is always invoked).

Prior to Linux 2.6.27, all access checks were of a single type. Since Linux 2.6.27, two access mode levels are distinguished:

PTRACE_MODE_READ

For "read" operations or other operations that are less dangerous, such as: get_robust_list(2); kcmp(2); reading /proc/[pid]/auxv, /proc/[pid]/environ, or /proc/[pid]/stat; or readlink(2) of a /proc/[pid]/ns/* file.

PTRACE_MODE_ATTACH

For "write" operations, or other operations that are more dangerous, such as: ptrace attaching (PTRACE_ATTACH) to another process or calling process_vm_writev(2). (PTRACE_MODE_ATTACH was effectively the default be? fore Linux 2.6.27.)

Since Linux 4.5, the above access mode checks are combined (ORed) with one of the following modifiers:

PTRACE_MODE_FSCREDS

Use the caller's filesystem UID and GID (see credentials(7)) or effective capabilities for LSM checks.

PTRACE_MODE_REALCREDS

Use the caller's real UID and GID or permitted capabilities for LSM checks.

This was effectively the default before Linux 4.5.

Because combining one of the credential modifiers with one of the aforementioned access modes is typical, some macros are defined in the kernel sources for the com? binations:

PTRACE_MODE_READ_FSCREDS

Defined as PTRACE_MODE_READ | PTRACE_MODE_FSCREDS.

PTRACE_MODE_READ_REALCREDS

Defined as PTRACE_MODE_READ | PTRACE_MODE_REALCREDS.

PTRACE_MODE_ATTACH_FSCREDS

Defined as PTRACE_MODE_ATTACH | PTRACE_MODE_FSCREDS.

PTRACE_MODE_ATTACH_REALCREDS

Defined as PTRACE_MODE_ATTACH | PTRACE_MODE_REALCREDS.

One further modifier can be ORed with the access mode:

PTRACE_MODE_NOAUDIT (since Linux 3.3)

Don't audit this access mode check. This modifier is employed for ptrace access mode checks (such as checks when reading /proc/[pid]/stat) that merely cause the output to be filtered or sanitized, rather than causing an error to be returned to the caller. In these cases, accessing the file is not a security violation and there is no reason to generate a security audit record. This modifier suppresses the generation of such an audit record for the particular access check.

Note that all of the PTRACE_MODE_* constants described in this subsection are ker? nel-internal, and not visible to user space. The constant names are mentioned here in order to label the various kinds of ptrace access mode checks that are performed for various system calls and accesses to various pseudofiles (e.g., under /proc). These names are used in other manual pages to provide a simple shorthand for label? ing the different kernel checks.

The algorithm employed for ptrace access mode checking determines whether the call? ing process is allowed to perform the corresponding action on the target process. (In the case of opening /proc/[pid] files, the "calling process" is the one opening the file, and the process with the corresponding PID is the "target process".) The algorithm is as follows:

1. If the calling thread and the target thread are in the same thread group, access

is always allowed.

2. If the access mode specifies PTRACE_MODE_FSCREDS, then, for the check in the next step, employ the caller's filesystem UID and GID. (As noted in creden? tials(7), the filesystem UID and GID almost always have the same values as the corresponding effective IDs.)

   Otherwise, the access mode specifies PTRACE_MODE_REALCREDS, so use the caller's real UID and GID for the checks in the next step. (Most APIs that check the caller's UID and GID use the effective IDs. For historical reasons, the PTRACE_MODE_REALCREDS check uses the real IDs instead.)

3. Deny access if neither of the following is true:

   ? The real, effective, and saved-set user IDs of the target match the caller's user ID, and the real, effective, and saved-set group IDs of the target match the caller's group ID.

   ? The caller has the CAP_SYS_PTRACE capability in the user namespace of the tar? get.

4. Deny access if the target process "dumpable" attribute has a value other than 1 (SUID_DUMP_USER; see the discussion of PR_SET_DUMPABLE in prctl(2)), and the caller does not have the CAP_SYS_PTRACE capability in the user namespace of the target process.

5. The kernel LSM security_ptrace_access_check() interface is invoked to see if ptrace access is permitted. The results depend on the LSM(s). The implementa? tion of this interface in the commoncap LSM performs the following steps:

   a) If the access mode includes PTRACE_MODE_FSCREDS, then use the caller's effec? tive capability set in the following check; otherwise (the access mode speci? fies PTRACE_MODE_REALCREDS, so) use the caller's permitted capability set.

   b) Deny access if neither of the following is true:

      ? The caller and the target process are in the same user namespace, and the caller's capabilities are a superset of the target process's permitted ca? pabilities.

      ? The caller has the CAP_SYS_PTRACE capability in the target process's user namespace.

      Note that the commoncap LSM does not distinguish between PTRACE_MODE_READ and PTRACE_MODE_ATTACH.

6. If access has not been denied by any of the preceding steps, then access is al?
lowed.

/proc/sys/kernel/yama/ptrace_scope

On systems with the Yama Linux Security Module (LSM) installed (i.e., the kernel was configured with CONFIG_SECURITY_YAMA), the /proc/sys/kernel/yama/ptrace_scope file (available since Linux 3.4) can be used to restrict the ability to trace a process with ptrace() (and thus also the ability to use tools such as strace(1) and gdb(1)). The goal of such restrictions is to prevent attack escalation whereby a compromised process can ptrace-attach to other sensitive processes (e.g., a GPG agent or an SSH session) owned by the user in order to gain additional credentials that may exist in memory and thus expand the scope of the attack.

More precisely, the Yama LSM limits two types of operations:

* Any operation that performs a ptrace access mode PTRACE_MODE_ATTACH check?for example, ptrace() PTRACE_ATTACH. (See the "Ptrace access mode checking" discus?sion above.)
* ptrace() PTRACE_TRACEME.

A process that has the CAP_SYS_PTRACE capability can update the /proc/sys/ker?nel/yama/ptrace_scope file with one of the following values:

0 ("classic ptrace permissions")

No additional restrictions on operations that perform PTRACE_MODE_ATTACH checks (beyond those imposed by the commoncap and other LSMs).

The use of PTRACE_TRACEME is unchanged.

1 ("restricted ptrace") [default value]

When performing an operation that requires a PTRACE_MODE_ATTACH check, the calling process must either have the CAP_SYS_PTRACE capability in the user namespace of the target process or it must have a predefined relationship with the target process. By default, the predefined relationship is that the target process must be a descendant of the caller.

A target process can employ the prctl(2) PR_SET_PTRACER operation to declare an additional PID that is allowed to perform PTRACE_MODE_ATTACH operations on the target. See the kernel source file Documentation/ad?min-guide/LSM/Yama.rst (or Documentation/security/Yama.txt before Linux 4.13) for further details.

The use of PTRACE_TRACEME is unchanged.

2 ("admin-only attach")

Only processes with the CAP_SYS_PTRACE capability in the user namespace of the target process may perform PTRACE_MODE_ATTACH operations or trace chil? dren that employ PTRACE_TRACEME.

3 ("no attach")

No process may perform PTRACE_MODE_ATTACH operations or trace children that employ PTRACE_TRACEME.

Once this value has been written to the file, it cannot be changed.

With respect to values 1 and 2, note that creating a new user namespace effectively removes the protection offered by Yama. This is because a process in the parent user namespace whose effective UID matches the UID of the creator of a child name? space has all capabilities (including CAP_SYS_PTRACE) when performing operations within the child user namespace (and further-removed descendants of that name? space). Consequently, when a process tries to use user namespaces to sandbox it? self, it inadvertently weakens the protections offered by the Yama LSM.

C library/kernel differences

At the system call level, the PTRACE_PEEKTEXT, PTRACE_PEEKDATA, and PTRACE_PEEKUSER requests have a different API: they store the result at the address specified by the data parameter, and the return value is the error flag. The glibc wrapper function provides the API given in DESCRIPTION above, with the result being re? turned via the function return value.

BUGS

On hosts with 2.6 kernel headers, PTRACE_SETOPTIONS is declared with a different value than the one for 2.4. This leads to applications compiled with 2.6 kernel headers failing when run on 2.4 kernels. This can be worked around by redefining PTRACE_SETOPTIONS to PTRACE_OLDSETOPTIONS, if that is defined.

Group-stop notifications are sent to the tracer, but not to real parent. Last con? firmed on 2.6.38.6.

If a thread group leader is traced and exits by calling _exit(2), a PTRACE_EVENT_EXIT stop will happen for it (if requested), but the subsequent WIFEX? ITED notification will not be delivered until all other threads exit. As explained above, if one of other threads calls execve(2), the death of the thread group

leader will never be reported. If the execed thread is not traced by this tracer, the tracer will never know that execve(2) happened. One possible workaround is to PTRACE_DETACH the thread group leader instead of restarting it in this case. Last confirmed on 2.6.38.6.

A SIGKILL signal may still cause a PTRACE_EVENT_EXIT stop before actual signal death. This may be changed in the future; SIGKILL is meant to always immediately kill tasks even under ptrace. Last confirmed on Linux 3.13.

Some system calls return with EINTR if a signal was sent to a tracee, but delivery was suppressed by the tracer. (This is very typical operation: it is usually done by debuggers on every attach, in order to not introduce a bogus SIGSTOP). As of Linux 3.2.9, the following system calls are affected (this list is likely incom? plete): epoll_wait(2), and read(2) from an inotify(7) file descriptor. The usual symptom of this bug is that when you attach to a quiescent process with the command

```
strace -p <process-ID>
```

then, instead of the usual and expected one-line output such as

```
restart_syscall(<... resuming interrupted call ...>_
```

or

```
select(6, [5], NULL, [5], NULL_
```

('_' denotes the cursor position), you observe more than one line. For example:

```
clock_gettime(CLOCK_MONOTONIC, {15370, 690928118}) = 0
epoll_wait(4,_
```

What is not visible here is that the process was blocked in epoll_wait(2) before strace(1) has attached to it. Attaching caused epoll_wait(2) to return to user space with the error EINTR. In this particular case, the program reacted to EINTR by checking the current time, and then executing epoll_wait(2) again. (Programs which do not expect such "stray" EINTR errors may behave in an unintended way upon an strace(1) attach.)

Contrary to the normal rules, the glibc wrapper for ptrace() can set errno to zero.

SEE ALSO

gdb(1), ltrace(1), strace(1), clone(2), execve(2), fork(2), gettid(2), prctl(2), seccomp(2), sigaction(2), tgkill(2), vfork(2), waitpid(2), exec(3), capabili? ties(7), signal(7)

COLOPHON

This page is part of release 5.05 of the Linux man-pages project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.