Red Hat Enterprise Linux Release 9.2 Manual Pages on 'seccomp.2' command

$ man seccomp.2

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

NAME

seccomp - operate on Secure Computing state of the process

SYNOPSIS

#include <linux/seccomp.h>

#include <linux/filter.h>

#include <linux/audit.h>

#include <linux/signal.h>

#include <sys/ptrace.h>

int seccomp(unsigned int operation, unsigned int flags, void *args);

DESCRIPTION

The seccomp() system call operates on the Secure Computing (seccomp)

state of the calling process.

Currently, Linux supports the following operation values:

SECCOMP_SET_MODE_STRICT

The only system calls that the calling thread is permitted to

make are read(2), write(2), _exit(2) (but not exit_group(2)),

and sigreturn(2). Other system calls result in the delivery of

a SIGKILL signal. Strict secure computing mode is useful for

number-crunching applications that may need to execute untrusted

byte code, perhaps obtained by reading from a pipe or socket.

Note that although the calling thread can no longer call sig?

procmask(2), it can use sigreturn(2) to block all signals apart

from SIGKILL and SIGSTOP. This means that alarm(2) (for exam?
ple) is not sufficient for restricting the process's execution
time. Instead, to reliably terminate the process, SIGKILL must
be used. This can be done by using timer_create(2) with
SIGEV_SIGNAL and sigev_signo set to SIGKILL, or by using setr?
limit(2) to set the hard limit for RLIMIT_CPU.

This operation is available only if the kernel is configured
with CONFIG_SECCOMP enabled.

The value of flags must be 0, and args must be NULL.

This operation is functionally identical to the call:

```
prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT);
```

SECCOMP_SET_MODE_FILTER

The system calls allowed are defined by a pointer to a Berkeley
Packet Filter (BPF) passed via args. This argument is a pointer
to a struct sock_fprog; it can be designed to filter arbitrary
system calls and system call arguments. If the filter is in?
valid, seccomp() fails, returning EINVAL in errno.

If fork(2) or clone(2) is allowed by the filter, any child pro?
cesses will be constrained to the same system call filters as
the parent. If execve(2) is allowed, the existing filters will
be preserved across a call to execve(2).

In order to use the SECCOMP_SET_MODE_FILTER operation, either
the calling thread must have the CAP_SYS_ADMIN capability in its
user namespace, or the thread must already have the no_new_privs
bit set. If that bit was not already set by an ancestor of this
thread, the thread must make the following call:

```
prctl(PR_SET_NO_NEW_PRIVS, 1);
```

Otherwise, the SECCOMP_SET_MODE_FILTER operation fails and re?
turns EACCES in errno. This requirement ensures that an unpriv?
ileged process cannot apply a malicious filter and then invoke a
set-user-ID or other privileged program using execve(2), thus
potentially compromising that program. (Such a malicious filter
might, for example, cause an attempt to use setuid(2) to set the

caller's user IDs to nonzero values to instead return 0 without
actually making the system call. Thus, the program might be
tricked into retaining superuser privileges in circumstances
where it is possible to influence it to do dangerous things be?
cause it did not actually drop privileges.)

If prctl(2) or seccomp() is allowed by the attached filter, fur?
ther filters may be added. This will increase evaluation time,
but allows for further reduction of the attack surface during
execution of a thread.

The SECCOMP_SET_MODE_FILTER operation is available only if the
kernel is configured with CONFIG_SECCOMP_FILTER enabled.

When flags is 0, this operation is functionally identical to the
call:

    prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, args);

The recognized flags are:

SECCOMP_FILTER_FLAG_TSYNC

When adding a new filter, synchronize all other threads
of the calling process to the same seccomp filter tree.
A "filter tree" is the ordered list of filters attached
to a thread. (Attaching identical filters in separate
seccomp() calls results in different filters from this
perspective.)

If any thread cannot synchronize to the same filter tree,
the call will not attach the new seccomp filter, and will
fail, returning the first thread ID found that cannot
synchronize. Synchronization will fail if another thread
in the same process is in SECCOMP_MODE_STRICT or if it
has attached new seccomp filters to itself, diverging
from the calling thread's filter tree.

SECCOMP_FILTER_FLAG_LOG (since Linux 4.14)

All filter return actions except SECCOMP_RET_ALLOW should
be logged. An administrator may override this filter
flag by preventing specific actions from being logged via

the /proc/sys/kernel/seccomp/actions_logged file.

SECCOMP_FILTER_FLAG_SPEC_ALLOW (since Linux 4.17)

Disable Speculative Store Bypass mitigation.

SECCOMP_GET_ACTION_AVAIL (since Linux 4.14)

Test to see if an action is supported by the kernel. This oper?
ation is helpful to confirm that the kernel knows of a more re?
cently added filter return action since the kernel treats all
unknown actions as SECCOMP_RET_KILL_PROCESS.

The value of flags must be 0, and args must be a pointer to an
unsigned 32-bit filter return action.

Filters

When adding filters via SECCOMP_SET_MODE_FILTER, args points to a fil?
ter program:

```
struct sock_fprog {
    unsigned short      len;    /* Number of BPF instructions */
    struct sock_filter *filter; /* Pointer to array of
                                   BPF instructions */
};
```

Each program must contain one or more BPF instructions:

```
struct sock_filter {            /* Filter block */
    __u16 code;                 /* Actual filter code */
    __u8  jt;                   /* Jump true */
    __u8  jf;                   /* Jump false */
    __u32 k;                    /* Generic multiuse field */
};
```

When executing the instructions, the BPF program operates on the system
call information made available (i.e., use the BPF_ABS addressing mode)
as a (read-only) buffer of the following form:

```
struct seccomp_data {
    int   nr;                   /* System call number */
    __u32 arch;                 /* AUDIT_ARCH_* value
                                   (see <linux/audit.h>) */
    __u64 instruction_pointer;  /* CPU instruction pointer */
```

```
    __u64 args[6];              /* Up to 6 system call arguments */
};
```

Because numbering of system calls varies between architectures and some architectures (e.g., x86-64) allow user-space code to use the calling conventions of multiple architectures (and the convention being used may vary over the life of a process that uses execve(2) to execute bi? naries that employ the different conventions), it is usually necessary to verify the value of the arch field.

It is strongly recommended to use an allow-list approach whenever pos? sible because such an approach is more robust and simple. A deny-list will have to be updated whenever a potentially dangerous system call is added (or a dangerous flag or option if those are deny-listed), and it is often possible to alter the representation of a value without alter? ing its meaning, leading to a deny-list bypass. See also Caveats be? low.

The arch field is not unique for all calling conventions. The x86-64 ABI and the x32 ABI both use AUDIT_ARCH_X86_64 as arch, and they run on the same processors. Instead, the mask __X32_SYSCALL_BIT is used on the system call number to tell the two ABIs apart.

This means that a policy must either deny all syscalls with __X32_SYSCALL_BIT or it must recognize syscalls with and without __X32_SYSCALL_BIT set. A list of system calls to be denied based on nr that does not also contain nr values with __X32_SYSCALL_BIT set can be bypassed by a malicious program that sets __X32_SYSCALL_BIT.

Additionally, kernels prior to Linux 5.4 incorrectly permitted nr in the ranges 512-547 as well as the corresponding non-x32 syscalls ORed with __X32_SYSCALL_BIT. For example, nr == 521 and nr == (101 | __X32_SYSCALL_BIT) would result in invocations of ptrace(2) with poten? tially confused x32-vs-x86_64 semantics in the kernel. Policies in? tended to work on kernels before Linux 5.4 must ensure that they deny or otherwise correctly handle these system calls. On Linux 5.4 and newer, such system calls will fail with the error ENOSYS, without doing anything.

The instruction_pointer field provides the address of the machine-lan?
guage instruction that performed the system call. This might be useful
in conjunction with the use of /proc/[pid]/maps to perform checks based
on which region (mapping) of the program made the system call. (Proba?
bly, it is wise to lock down the mmap(2) and mprotect(2) system calls
to prevent the program from subverting such checks.)

When checking values from args, keep in mind that arguments are often
silently truncated before being processed, but after the seccomp check.
For example, this happens if the i386 ABI is used on an x86-64 kernel:
although the kernel will normally not look beyond the 32 lowest bits of
the arguments, the values of the full 64-bit registers will be present
in the seccomp data. A less surprising example is that if the x86-64
ABI is used to perform a system call that takes an argument of type
int, the more-significant half of the argument register is ignored by
the system call, but visible in the seccomp data.

A seccomp filter returns a 32-bit value consisting of two parts: the
most significant 16 bits (corresponding to the mask defined by the con?
stant SECCOMP_RET_ACTION_FULL) contain one of the "action" values
listed below; the least significant 16-bits (defined by the constant
SECCOMP_RET_DATA) are "data" to be associated with this return value.

If multiple filters exist, they are all executed, in reverse order of
their addition to the filter tree?that is, the most recently installed
filter is executed first. (Note that all filters will be called even
if one of the earlier filters returns SECCOMP_RET_KILL. This is done
to simplify the kernel code and to provide a tiny speed-up in the exe?
cution of sets of filters by avoiding a check for this uncommon case.)

The return value for the evaluation of a given system call is the
first-seen action value of highest precedence (along with its accompa?
nying data) returned by execution of all of the filters.

In decreasing order of precedence, the action values that may be re?
turned by a seccomp filter are:

SECCOMP_RET_KILL_PROCESS (since Linux 4.14)

This value results in immediate termination of the process, with a core dump. The system call is not executed. By contrast with SECCOMP_RET_KILL_THREAD below, all threads in the thread group are terminated. (For a discussion of thread groups, see the de? scription of the CLONE_THREAD flag in clone(2).)

The process terminates as though killed by a SIGSYS signal. Even if a signal handler has been registered for SIGSYS, the handler will be ignored in this case and the process always ter? minates. To a parent process that is waiting on this process (using waitpid(2) or similar), the returned wstatus will indi? cate that its child was terminated as though by a SIGSYS signal.

SECCOMP_RET_KILL_THREAD (or SECCOMP_RET_KILL)

This value results in immediate termination of the thread that made the system call. The system call is not executed. Other threads in the same thread group will continue to execute.

The thread terminates as though killed by a SIGSYS signal. See SECCOMP_RET_KILL_PROCESS above.

Before Linux 4.11, any process terminated in this way would not trigger a coredump (even though SIGSYS is documented in sig? nal(7) as having a default action of termination with a core dump). Since Linux 4.11, a single-threaded process will dump core if terminated in this way.

With the addition of SECCOMP_RET_KILL_PROCESS in Linux 4.14, SECCOMP_RET_KILL_THREAD was added as a synonym for SEC? COMP_RET_KILL, in order to more clearly distinguish the two ac? tions.

Note: the use of SECCOMP_RET_KILL_THREAD to kill a single thread in a multithreaded process is likely to leave the process in a permanently inconsistent and possibly corrupt state.

SECCOMP_RET_TRAP

This value results in the kernel sending a thread-directed SIGSYS signal to the triggering thread. (The system call is not executed.) Various fields will be set in the siginfo_t struc? ture (see sigaction(2)) associated with signal:

* si_signo will contain SIGSYS.

* si_call_addr will show the address of the system call in?
  struction.

* si_syscall and si_arch will indicate which system call was
  attempted.

* si_code will contain SYS_SECCOMP.

* si_errno will contain the SECCOMP_RET_DATA portion of the
  filter return value.

The program counter will be as though the system call happened (i.e., the program counter will not point to the system call in? struction). The return value register will contain an architec? ture-dependent value; if resuming execution, set it to something appropriate for the system call. (The architecture dependency is because replacing it with ENOSYS could overwrite some useful information.)

SECCOMP_RET_ERRNO

This value results in the SECCOMP_RET_DATA portion of the fil? ter's return value being passed to user space as the errno value without executing the system call.

SECCOMP_RET_TRACE

When returned, this value will cause the kernel to attempt to notify a ptrace(2)-based tracer prior to executing the system call. If there is no tracer present, the system call is not ex? ecuted and returns a failure status with errno set to ENOSYS.

A tracer will be notified if it requests PTRACE_O_TRACESECCOMP using ptrace(PTRACE_SETOPTIONS). The tracer will be notified of a PTRACE_EVENT_SECCOMP and the SECCOMP_RET_DATA portion of the filter's return value will be available to the tracer via PTRACE_GETEVENTMSG.

The tracer can skip the system call by changing the system call number to -1. Alternatively, the tracer can change the system call requested by changing the system call to a valid system call number. If the tracer asks to skip the system call, then the system call will appear to return the value that the tracer puts in the return value register.

Before kernel 4.8, the seccomp check will not be run again after the tracer is notified. (This means that, on older kernels, seccomp-based sandboxes must not allow use of ptrace(2)?even of other sandboxed processes?without extreme care; ptracers can use this mechanism to escape from the seccomp sandbox.)

Note that a tracer process will not be notified if another fil?ter returns an action value with a precedence greater than SEC?COMP_RET_TRACE.

SECCOMP_RET_LOG (since Linux 4.14)

This value results in the system call being executed after the filter return action is logged. An administrator may override the logging of this action via the /proc/sys/kernel/seccomp/ac?tions_logged file.

SECCOMP_RET_ALLOW

This value results in the system call being executed.

If an action value other than one of the above is specified, then the filter action is treated as either SECCOMP_RET_KILL_PROCESS (since Linux 4.14) or SECCOMP_RET_KILL_THREAD (in Linux 4.13 and earlier).

/proc interfaces

The files in the directory /proc/sys/kernel/seccomp provide additional seccomp information and configuration:

actions_avail (since Linux 4.14)

A read-only ordered list of seccomp filter return actions in string form. The ordering, from left-to-right, is in decreasing order of precedence. The list represents the set of seccomp filter return actions supported by the kernel.

actions_logged (since Linux 4.14)

A read-write ordered list of seccomp filter return actions that are allowed to be logged. Writes to the file do not need to be in ordered form but reads from the file will be ordered in the same way as the actions_avail file.

It is important to note that the value of actions_logged does not prevent certain filter return actions from being logged when the audit subsystem is configured to audit a task. If the ac? tion is not found in the actions_logged file, the final decision on whether to audit the action for that task is ultimately left up to the audit subsystem to decide for all filter return ac? tions other than SECCOMP_RET_ALLOW.

The "allow" string is not accepted in the actions_logged file as it is not possible to log SECCOMP_RET_ALLOW actions. Attempting to write "allow" to the file will fail with the error EINVAL.

Audit logging of seccomp actions

Since Linux 4.14, the kernel provides the facility to log the actions returned by seccomp filters in the audit log. The kernel makes the de? cision to log an action based on the action type, whether or not the action is present in the actions_logged file, and whether kernel audit? ing is enabled (e.g., via the kernel boot option audit=1). The rules are as follows:

* If the action is SECCOMP_RET_ALLOW, the action is not logged.
* Otherwise, if the action is either SECCOMP_RET_KILL_PROCESS or SEC? COMP_RET_KILL_THREAD, and that action appears in the actions_logged file, the action is logged.
* Otherwise, if the filter has requested logging (the SECCOMP_FIL? TER_FLAG_LOG flag) and the action appears in the actions_logged file, the action is logged.
* Otherwise, if kernel auditing is enabled and the process is being audited (autrace(8)), the action is logged.
* Otherwise, the action is not logged.

RETURN VALUE

On success, seccomp() returns 0. On error, if SECCOMP_FIL? TER_FLAG_TSYNC was used, the return value is the ID of the thread that caused the synchronization failure. (This ID is a kernel thread ID of the type returned by clone(2) and gettid(2).) On other errors, -1 is returned, and errno is set to indicate the cause of the error.

ERRORS

seccomp() can fail for the following reasons:

EACCES The caller did not have the CAP_SYS_ADMIN capability in its user namespace, or had not set no_new_privs before using SEC?COMP_SET_MODE_FILTER.

EFAULT args was not a valid address.

EINVAL operation is unknown or is not supported by this kernel version or configuration.

EINVAL The specified flags are invalid for the given operation.

EINVAL operation included BPF_ABS, but the specified offset was not aligned to a 32-bit boundary or exceeded sizeof(struct sec?comp_data).

EINVAL A secure computing mode has already been set, and operation dif?fers from the existing setting.

EINVAL operation specified SECCOMP_SET_MODE_FILTER, but the filter pro?gram pointed to by args was not valid or the length of the fil?ter program was zero or exceeded BPF_MAXINSNS (4096) instruc?tions.

ENOMEM Out of memory.

ENOMEM The total length of all filter programs attached to the calling thread would exceed MAX_INSNS_PER_PATH (32768) instructions. Note that for the purposes of calculating this limit, each al?ready existing filter program incurs an overhead penalty of 4 instructions.

EOPNOTSUPP

operation specified SECCOMP_GET_ACTION_AVAIL, but the kernel does not support the filter return action specified by args.

ESRCH Another thread caused a failure during thread sync, but its ID could not be determined.

VERSIONS

The seccomp() system call first appeared in Linux 3.17.

CONFORMING TO

The seccomp() system call is a nonstandard Linux extension.

NOTES

Rather than hand-coding seccomp filters as shown in the example below, you may prefer to employ the libseccomp library, which provides a front-end for generating seccomp filters.

The Seccomp field of the /proc/[pid]/status file provides a method of viewing the seccomp mode of a process; see proc(5).

seccomp() provides a superset of the functionality provided by the prctl(2) PR_SET_SECCOMP operation (which does not support flags).

Since Linux 4.4, the ptrace(2) PTRACE_SECCOMP_GET_FILTER operation can be used to dump a process's seccomp filters.

Architecture support for seccomp BPF

Architecture support for seccomp BPF filtering is available on the fol?
lowing architectures:

* x86-64, i386, x32 (since Linux 3.5)
* ARM (since Linux 3.8)
* s390 (since Linux 3.8)
* MIPS (since Linux 3.16)
* ARM-64 (since Linux 3.19)
* PowerPC (since Linux 4.3)
* Tile (since Linux 4.3)
* PA-RISC (since Linux 4.6)

Caveats

There are various subtleties to consider when applying seccomp filters to a program, including the following:

* Some traditional system calls have user-space implementations in the vdso(7) on many architectures. Notable examples include clock_get?
time(2), gettimeofday(2), and time(2). On such architectures, sec?
comp filtering for these system calls will have no effect. (How?
ever, there are cases where the vdso(7) implementations may fall back to invoking the true system call, in which case seccomp filters would see the system call.)

* Seccomp filtering is based on system call numbers. However, appli?
cations typically do not directly invoke system calls, but instead

call wrapper functions in the C library which in turn invoke the system calls. Consequently, one must be aware of the following:

? The glibc wrappers for some traditional system calls may actually employ system calls with different names in the kernel. For ex? ample, the exit(2) wrapper function actually employs the exit_group(2) system call, and the fork(2) wrapper function actu? ally calls clone(2).

? The behavior of wrapper functions may vary across architectures, according to the range of system calls provided on those archi? tectures. In other words, the same wrapper function may invoke different system calls on different architectures.

? Finally, the behavior of wrapper functions can change across glibc versions. For example, in older versions, the glibc wrap? per function for open(2) invoked the system call of the same name, but starting in glibc 2.26, the implementation switched to calling openat(2) on all architectures.

The consequence of the above points is that it may be necessary to fil? ter for a system call other than might be expected. Various manual pages in Section 2 provide helpful details about the differences be? tween wrapper functions and the underlying system calls in subsections entitled C library/kernel differences.

Furthermore, note that the application of seccomp filters even risks causing bugs in an application, when the filters cause unexpected fail? ures for legitimate operations that the application might need to per? form. Such bugs may not easily be discovered when testing the seccomp filters if the bugs occur in rarely used application code paths.

Seccomp-specific BPF details

Note the following BPF details specific to seccomp filters:

* The BPF_H and BPF_B size modifiers are not supported: all operations must load and store (4-byte) words (BPF_W).

* To access the contents of the seccomp_data buffer, use the BPF_ABS addressing mode modifier.

* The BPF_LEN addressing mode modifier yields an immediate mode oper?

and whose value is the size of the seccomp_data buffer.

EXAMPLES

The program below accepts four or more arguments. The first three ar? guments are a system call number, a numeric architecture identifier, and an error number. The program uses these values to construct a BPF filter that is used at run time to perform the following checks:

[1] If the program is not running on the specified architecture, the BPF filter causes system calls to fail with the error ENOSYS.

[2] If the program attempts to execute the system call with the speci? fied number, the BPF filter causes the system call to fail, with errno being set to the specified error number.

The remaining command-line arguments specify the pathname and addi? tional arguments of a program that the example program should attempt to execute using execv(3) (a library function that employs the ex? ecve(2) system call). Some example runs of the program are shown be? low.

First, we display the architecture that we are running on (x86-64) and then construct a shell function that looks up system call numbers on this architecture:

```
$ uname -m
x86_64
$ syscall_nr() {
    cat /usr/src/linux/arch/x86/syscalls/syscall_64.tbl | \
    awk '$2 != "x32" && $3 == "'$1'" { print $1 }'
}
```

When the BPF filter rejects a system call (case [2] above), it causes the system call to fail with the error number specified on the command line. In the experiments shown here, we'll use error number 99:

```
$ errno 99
EADDRNOTAVAIL 99 Cannot assign requested address
```

In the following example, we attempt to run the command whoami(1), but the BPF filter rejects the execve(2) system call, so that the command is not even executed:

```
$ syscall_nr execve
59
$ ./a.out
Usage: ./a.out <syscall_nr> <arch> <errno> <prog> [<args>]
Hint for <arch>: AUDIT_ARCH_I386: 0x40000003
                 AUDIT_ARCH_X86_64: 0xC000003E
$ ./a.out 59 0xC000003E 99 /bin/whoami
execv: Cannot assign requested address
```

In the next example, the BPF filter rejects the write(2) system call, so that, although it is successfully started, the whoami(1) command is not able to write output:

```
$ syscall_nr write
1
$ ./a.out 1 0xC000003E 99 /bin/whoami
```

In the final example, the BPF filter rejects a system call that is not used by the whoami(1) command, so it is able to successfully execute and produce output:

```
$ syscall_nr preadv
295
$ ./a.out 295 0xC000003E 99 /bin/whoami
cecilia
```

Program source

```
#include <errno.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <linux/audit.h>
#include <linux/filter.h>
#include <linux/seccomp.h>
#include <sys/prctl.h>
#define X32_SYSCALL_BIT 0x40000000
#define ARRAY_SIZE(arr) (sizeof(arr) / sizeof((arr)[0]))
```

```
static int
install_filter(int syscall_nr, int t_arch, int f_errno)
{
    unsigned int upper_nr_limit = 0xffffffff;
    /* Assume that AUDIT_ARCH_X86_64 means the normal x86-64 ABI
       (in the x32 ABI, all system calls have bit 30 set in the
       'nr' field, meaning the numbers are >= X32_SYSCALL_BIT) */
    if (t_arch == AUDIT_ARCH_X86_64)
        upper_nr_limit = X32_SYSCALL_BIT - 1;
    struct sock_filter filter[] = {
        /* [0] Load architecture from 'seccomp_data' buffer into
               accumulator */
        BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
                 (offsetof(struct seccomp_data, arch))),
        /* [1] Jump forward 5 instructions if architecture does not
               match 't_arch' */
        BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, t_arch, 0, 5),
        /* [2] Load system call number from 'seccomp_data' buffer into
               accumulator */
        BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
                 (offsetof(struct seccomp_data, nr))),
        /* [3] Check ABI - only needed for x86-64 in deny-list use
               cases.  Use BPF_JGT instead of checking against the bit
               mask to avoid having to reload the syscall number. */
        BPF_JUMP(BPF_JMP | BPF_JGT | BPF_K, upper_nr_limit, 3, 0),
        /* [4] Jump forward 1 instruction if system call number
               does not match 'syscall_nr' */
        BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, syscall_nr, 0, 1),
        /* [5] Matching architecture and system call: don't execute
           the system call, and return 'f_errno' in 'errno' */
        BPF_STMT(BPF_RET | BPF_K,
                 SECCOMP_RET_ERRNO | (f_errno & SECCOMP_RET_DATA)),
        /* [6] Destination of system call number mismatch: allow other
```

```
                system calls */
        BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
        /* [7] Destination of architecture mismatch: kill process */
        BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL_PROCESS),
    };
    struct sock_fprog prog = {
        .len = ARRAY_SIZE(filter),
        .filter = filter,
    };
    if (seccomp(SECCOMP_SET_MODE_FILTER, 0, &prog)) {
        perror("seccomp");
        return 1;
    }
    return 0;
}
int
main(int argc, char **argv)
{
    if (argc < 5) {
        fprintf(stderr, "Usage: "
                "%s <syscall_nr> <arch> <errno> <prog> [<args>]\n"
                "Hint for <arch>: AUDIT_ARCH_I386: 0x%X\n"
                "                 AUDIT_ARCH_X86_64: 0x%X\n"
                "\n", argv[0], AUDIT_ARCH_I386, AUDIT_ARCH_X86_64);
        exit(EXIT_FAILURE);
    }
    if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0)) {
        perror("prctl");
        exit(EXIT_FAILURE);
    }
    if (install_filter(strtol(argv[1], NULL, 0),
                       strtol(argv[2], NULL, 0),
                       strtol(argv[3], NULL, 0)))
```

```
            exit(EXIT_FAILURE);

        execv(argv[4], &argv[4]);

        perror("execv");

        exit(EXIT_FAILURE);

    }
```

SEE ALSO

bpfc(1), strace(1), bpf(2), prctl(2), ptrace(2), sigaction(2), proc(5),

signal(7), socket(7)

Various pages from the libseccomp library, including: scmp_sys_re?

solver(1), seccomp_export_bpf(3), seccomp_init(3), seccomp_load(3), and

seccomp_rule_add(3).

The kernel source files Documentation/networking/filter.txt and Docu?

mentation/userspace-api/seccomp_filter.rst (or Documentation/prctl/sec?

comp_filter.txt before Linux 4.13).

McCanne, S. and Jacobson, V. (1992) The BSD Packet Filter: A New Archi?

tecture for User-level Packet Capture, Proceedings of the USENIX Winter

1993 Conference ?http://www.tcpdump.org/papers/bpf-usenix93.pdf?

COLOPHON

This page is part of release 5.10 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/.