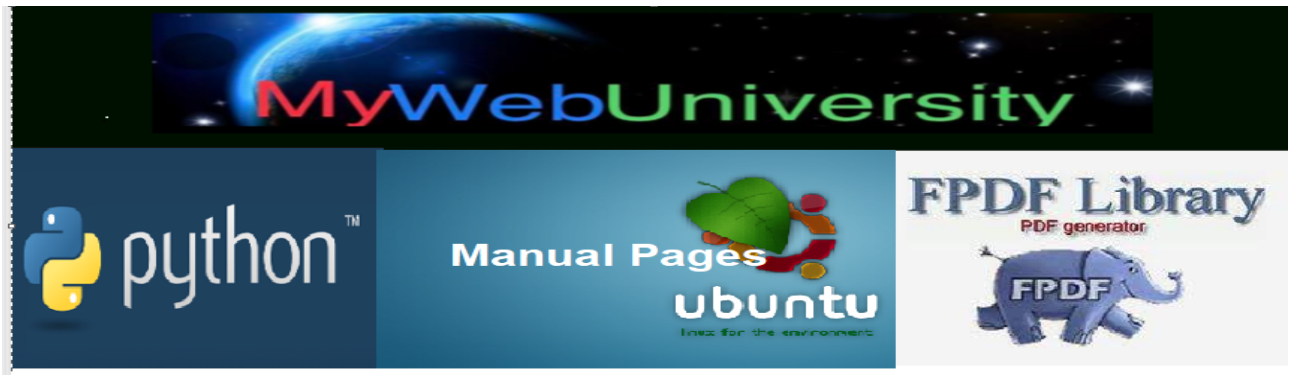

Full credit is given to the above companies including the OS that this PDF file was generated!

Linux Ubuntu 26.04 Manual Pages on command 'seccomp_unotify.2'

$ man seccomp_unotify.2

seccomp_unotify(2) System Calls Manual seccomp_unotify(2)

NAME

seccomp_unotify - Seccomp user-space notification mechanism

LIBRARY

Standard C library (libc, -lc)

SYNOPSIS

```
#include <linux/seccomp.h>
#include <linux/filter.h>
#include <linux/audit.h>
int seccomp(unsigned int operation, unsigned int flags, void *args);
#include <sys/ioctl.h>
int ioctl(int fd, SECCOMP_IOCTL_NOTIF_RECV,
          struct seccomp_notif *req);
int ioctl(int fd, SECCOMP_IOCTL_NOTIF_SEND,
          struct seccomp_notif_resp *resp);
int ioctl(int fd, SECCOMP_IOCTL_NOTIF_ID_VALID, __u64 *id);
int ioctl(int fd, SECCOMP_IOCTL_NOTIF_ADDFD,
          struct seccomp_notif_addfd *addfd);
```

DESCRIPTION

This page describes the user-space notification mechanism provided by the Secure Computing (seccomp) facility. As well as the use of the SECCOMP_FILTER_FLAG_NEW_LISTENER flag, the SECCOMP_RET_USER_NOTIF action value, and the SECCOMP_GET_NOTIF_SIZES operation described in seccomp(2), this mechanism involves the use of a number of related ioctl(2) operations (de?

scribed below).

Overview

In conventional usage of a seccomp filter, the decision about how to treat a system call is made by the filter itself. By contrast, the user-space notification mechanism allows the seccomp filter to delegate the handling of the system call to another user-space process. Note that this mechanism is explicitly not intended as a method implementing security pol? icy; see NOTES.

In the discussion that follows, the thread(s) on which the seccomp filter is installed is (are) referred to as the target, and the process that is notified by the user-space notifi? cation mechanism is referred to as the supervisor.

A suitably privileged supervisor can use the user-space notification mechanism to perform actions on behalf of the target. The advantage of the user-space notification mechanism is that the supervisor will usually be able to retrieve information about the target and the performed system call that the seccomp filter itself cannot. (A seccomp filter is limited in the information it can obtain and the actions that it can perform because it is running on a virtual machine inside the kernel.)

An overview of the steps performed by the target and the supervisor is as follows:

(1) The target establishes a seccomp filter in the usual manner, but with two differences:

? The seccomp(2) flags argument includes the flag SECCOMP_FILTER_FLAG_NEW_LISTENER. Consequently, the return value of the (successful) seccomp(2) call is a new "listen? ing" file descriptor that can be used to receive notifications. Only one "listen? ing" seccomp filter can be installed for a thread.

? In cases where it is appropriate, the seccomp filter returns the action value SEC? COMP_RET_USER_NOTIF. This return value will trigger a notification event.

(2) In order that the supervisor can obtain notifications using the listening file descrip? tor, (a duplicate of) that file descriptor must be passed from the target to the super? visor. One way in which this could be done is by passing the file descriptor over a UNIX domain socket connection between the target and the supervisor (using the SCM_RIGHTS ancillary message type described in unix(7)). Another way to do this is through the use of pidfd_getfd(2).

(3) The supervisor will receive notification events on the listening file descriptor. These events are returned as structures of type seccomp_notif. Because this structure and its size may evolve over kernel versions, the supervisor must first determine the

size of this structure using the seccomp(2) SECCOMP_GET_NOTIF_SIZES operation, which returns a structure of type seccomp_notif_sizes. The supervisor allocates a buffer of size seccomp_notif_sizes.seccomp_notif bytes to receive notification events. In addi? tion,the supervisor allocates another buffer of size seccomp_notif_sizes.seccomp_no? tif_resp bytes for the response (a struct seccomp_notif_resp structure) that it will provide to the kernel (and thus the target).

(4) The target then performs its workload, which includes system calls that will be con? trolled by the seccomp filter. Whenever one of these system calls causes the filter to return the SECCOMP_RET_USER_NOTIF action value, the kernel does not (yet) execute the system call; instead, execution of the target is temporarily blocked inside the kernel (in a sleep state that is interruptible by signals) and a notification event is gener? ated on the listening file descriptor.

(5) The supervisor can now repeatedly monitor the listening file descriptor for SEC? COMP_RET_USER_NOTIF-triggered events. To do this, the supervisor uses the SEC? COMP_IOCTL_NOTIF_RECV ioctl(2) operation to read information about a notification event; this operation blocks until an event is available. The operation returns a sec? comp_notif structure containing information about the system call that is being at? tempted by the target. (As described in NOTES, the file descriptor can also be moni? tored with select(2), poll(2), or epoll(7).)

(6) The seccomp_notif structure returned by the SECCOMP_IOCTL_NOTIF_RECV operation includes the same information (a seccomp_data structure) that was passed to the seccomp filter. This information allows the supervisor to discover the system call number and the argu? ments for the target's system call. In addition, the notification event contains the ID of the thread that triggered the notification and a unique cookie value that is used in subsequent SECCOMP_IOCTL_NOTIF_ID_VALID and SECCOMP_IOCTL_NOTIF_SEND operations. The information in the notification can be used to discover the values of pointer argu? ments for the target's system call. (This is something that can't be done from within a seccomp filter.) One way in which the supervisor can do this is to open the corre? sponding /proc/tid/mem file (see proc(5)) and read bytes from the location that corre? sponds to one of the pointer arguments whose value is supplied in the notification event. (The supervisor must be careful to avoid a race condition that can occur when doing this; see the description of the SECCOMP_IOCTL_NOTIF_ID_VALID ioctl(2) operation below.) In addition, the supervisor can access other system information that is visi?

ble in user space but which is not accessible from a seccomp filter.

(7) Having obtained information as per the previous step, the supervisor may then choose to perform an action in response to the target's system call (which, as noted above, is not executed when the seccomp filter returns the SECCOMP_RET_USER_NOTIF action value). One example use case here relates to containers. The target may be located inside a container where it does not have sufficient capabilities to mount a filesystem in the container's mount namespace. However, the supervisor may be a more privileged process that does have sufficient capabilities to perform the mount operation.

(8) The supervisor then sends a response to the notification. The information in this re? sponse is used by the kernel to construct a return value for the target's system call and provide a value that will be assigned to the errno variable of the target. The response is sent using the SECCOMP_IOCTL_NOTIF_SEND ioctl(2) operation, which is used to transmit a seccomp_notif_resp structure to the kernel. This structure includes a cookie value that the supervisor obtained in the seccomp_notif structure returned by the SECCOMP_IOCTL_NOTIF_RECV operation. This cookie value allows the kernel to asso? ciate the response with the target. This structure must include the cookie value that the supervisor obtained in the seccomp_notif structure returned by the SEC? COMP_IOCTL_NOTIF_RECV operation; the cookie allows the kernel to associate the response with the target.

(9) Once the notification has been sent, the system call in the target thread unblocks, re? turning the information that was provided by the supervisor in the notification re? sponse.

As a variation on the last two steps, the supervisor can send a response that tells the ker? nel that it should execute the target thread's system call; see the discussion of SEC? COMP_USER_NOTIF_FLAG_CONTINUE, below.

IOCTL OPERATIONS

The following ioctl(2) operations are supported by the seccomp user-space notification file descriptor. For each of these operations, the first (file descriptor) argument of ioctl(2) is the listening file descriptor returned by a call to seccomp(2) with the SECCOMP_FIL? TER_FLAG_NEW_LISTENER flag.

SECCOMP_IOCTL_NOTIF_RECV

The SECCOMP_IOCTL_NOTIF_RECV operation (available since Linux 5.0) is used to obtain a user-space notification event. If no such event is currently pending, the operation blocks until

an event occurs. The third ioctl(2) argument is a pointer to a structure of the following form which contains information about the event. This structure must be zeroed out before the call.

```
struct seccomp_notif {
    __u64  id;             /* Cookie */
    __u32  pid;            /* TID of target thread */
    __u32  flags;          /* Currently unused (0) */
    struct seccomp_data data;   /* See seccomp(2) */
};
```

The fields in this structure are as follows:

id This is a cookie for the notification. Each such cookie is guaranteed to be unique for the corresponding seccomp filter.

? The cookie can be used with the SECCOMP_IOCTL_NOTIF_ID_VALID ioctl(2) operation described below.

? When returning a notification response to the kernel, the supervisor must include the cookie value in the seccomp_notif_resp structure that is specified as the ar? gument of the SECCOMP_IOCTL_NOTIF_SEND operation.

pid This is the thread ID of the target thread that triggered the notification event.

flags This is a bit mask of flags providing further information on the event. In the cur? rent implementation, this field is always zero.

data This is a seccomp_data structure containing information about the system call that triggered the notification. This is the same structure that is passed to the seccomp filter. See seccomp(2) for details of this structure.

On success, this operation returns 0; on failure, -1 is returned, and errno is set to indi? cate the error. This operation can fail with the following errors:

EINVAL (since Linux 5.5)

The seccomp_notif structure that was passed to the call contained nonzero fields.

ENOENT The target thread was killed by a signal as the notification information was being generated, or the target's (blocked) system call was interrupted by a signal handler.

SECCOMP_IOCTL_NOTIF_ID_VALID

The SECCOMP_IOCTL_NOTIF_ID_VALID operation (available since Linux 5.0) is used to check that a notification ID returned by an earlier SECCOMP_IOCTL_NOTIF_RECV operation is still valid (i.e., that the target still exists and its system call is still blocked waiting for a re?

sponse).

The third ioctl(2) argument is a pointer to the cookie (id) returned by the SEC? COMP_IOCTL_NOTIF_RECV operation.

This operation is necessary to avoid race conditions that can occur when the pid returned by the SECCOMP_IOCTL_NOTIF_RECV operation terminates, and that process ID is reused by another process. An example of this kind of race is the following

(1) A notification is generated on the listening file descriptor. The returned seccomp_no? tif contains the TID of the target thread (in the pid field of the structure).

(2) The target terminates.

(3) Another thread or process is created on the system that by chance reuses the TID that was freed when the target terminated.

(4) The supervisor open(2)s the /proc/tid/mem file for the TID obtained in step 1, with the intention of (say) inspecting the memory location(s) that containing the argument(s) of the system call that triggered the notification in step 1.

In the above scenario, the risk is that the supervisor may try to access the memory of a process other than the target. This race can be avoided by following the call to open(2) with a SECCOMP_IOCTL_NOTIF_ID_VALID operation to verify that the process that generated the notification is still alive. (Note that if the target terminates after the latter step, a subsequent read(2) from the file descriptor may return 0, indicating end of file.)

See NOTES for a discussion of other cases where SECCOMP_IOCTL_NOTIF_ID_VALID checks must be performed.

On success (i.e., the notification ID is still valid), this operation returns 0. On failure (i.e., the notification ID is no longer valid), -1 is returned, and errno is set to ENOENT.

SECCOMP_IOCTL_NOTIF_SEND

The SECCOMP_IOCTL_NOTIF_SEND operation (available since Linux 5.0) is used to send a notifi? cation response back to the kernel. The third ioctl(2) argument of this structure is a pointer to a structure of the following form:

```
struct seccomp_notif_resp {
    __u64 id;          /* Cookie value */
    __s64 val;         /* Success return value */
    __s32 error;       /* 0 (success) or negative error number */
    __u32 flags;       /* See below */
};
```

The fields of this structure are as follows:

id This is the cookie value that was obtained using the SECCOMP_IOCTL_NOTIF_RECV opera? tion. This cookie value allows the kernel to correctly associate this response with the system call that triggered the user-space notification.

val This is the value that will be used for a spoofed success return for the target's system call; see below.

error This is the value that will be used as the error number (errno) for a spoofed error return for the target's system call; see below.

flags This is a bit mask that includes zero or more of the following flags:

SECCOMP_USER_NOTIF_FLAG_CONTINUE (since Linux 5.5)

Tell the kernel to execute the target's system call.

Two kinds of response are possible:

? A response to the kernel telling it to execute the target's system call. In this case, the flags field includes SECCOMP_USER_NOTIF_FLAG_CONTINUE and the error and val fields must be zero.

This kind of response can be useful in cases where the supervisor needs to do deeper analysis of the target's system call than is possible from a seccomp filter (e.g., exam? ining the values of pointer arguments), and, having decided that the system call does not require emulation by the supervisor, the supervisor wants the system call to be executed normally in the target.

The SECCOMP_USER_NOTIF_FLAG_CONTINUE flag should be used with caution; see NOTES.

? A spoofed return value for the target's system call. In this case, the kernel does not execute the target's system call, instead causing the system call to return a spoofed value as specified by fields of the seccomp_notif_resp structure. The supervisor should set the fields of this structure as follows:

? flags does not contain SECCOMP_USER_NOTIF_FLAG_CONTINUE.

? error is set either to 0 for a spoofed "success" return or to a negative error number for a spoofed "failure" return. In the former case, the kernel causes the target's system call to return the value specified in the val field. In the latter case, the kernel causes the target's system call to return -1, and errno is assigned the negated error value.

? val is set to a value that will be used as the return value for a spoofed "success" return for the target's system call. The value in this field is ignored if the error

field contains a nonzero value.

On success, this operation returns 0; on failure, -1 is returned, and errno is set to indi?
cate the error. This operation can fail with the following errors:

EINPROGRESS

A response to this notification has already been sent.

EINVAL An invalid value was specified in the flags field.

EINVAL The flags field contained SECCOMP_USER_NOTIF_FLAG_CONTINUE, and the error or val
field was not zero.

ENOENT The blocked system call in the target has been interrupted by a signal handler or the
target has terminated.

SECCOMP_IOCTL_NOTIF_ADDFD

The SECCOMP_IOCTL_NOTIF_ADDFD operation (available since Linux 5.9) allows the supervisor to
install a file descriptor into the target's file descriptor table. Much like the use of
SCM_RIGHTS messages described in unix(7), this operation is semantically equivalent to du?
plicating a file descriptor from the supervisor's file descriptor table into the target's
file descriptor table.

The SECCOMP_IOCTL_NOTIF_ADDFD operation permits the supervisor to emulate a target system
call (such as socket(2) or openat(2)) that generates a file descriptor. The supervisor can
perform the system call that generates the file descriptor (and associated open file de?
scription) and then use this operation to allocate a file descriptor that refers to the same
open file description in the target. (For an explanation of open file descriptions, see
open(2).)

Once this operation has been performed, the supervisor can close its copy of the file de?
scriptor.

In the target, the received file descriptor is subject to the same Linux Security Module
(LSM) checks as are applied to a file descriptor that is received in an SCM_RIGHTS ancillary
message. If the file descriptor refers to a socket, it inherits the cgroup version 1 net?
work controller settings (classid and netprioidx) of the target.

The third ioctl(2) argument is a pointer to a structure of the following form:

```
struct seccomp_notif_addfd {
    __u64 id;          /* Cookie value */
    __u32 flags;       /* Flags */
    __u32 srcfd;       /* Local file descriptor number */
```

```
    __u32 newfd;        /* 0 or desired file descriptor
                           number in target */
    __u32 newfd_flags;  /* Flags to set on target file
                           descriptor */
};
```

The fields in this structure are as follows:

id This field should be set to the notification ID (cookie value) that was obtained via SECCOMP_IOCTL_NOTIF_RECV.

flags This field is a bit mask of flags that modify the behavior of the operation. Cur? rently, only one flag is supported:

SECCOMP_ADDFD_FLAG_SETFD

When allocating the file descriptor in the target, use the file descriptor number specified in the newfd field.

SECCOMP_ADDFD_FLAG_SEND (since Linux 5.14)

Perform the equivalent of SECCOMP_IOCTL_NOTIF_ADDFD plus SECCOMP_IOCTL_NO? TIF_SEND as an atomic operation. On successful invocation, the target process's errno will be 0 and the return value will be the file descriptor number that was allocated in the target. If allocating the file descriptor in the target fails, the target's system call continues to be blocked until a successful response is sent.

srcfd This field should be set to the number of the file descriptor in the supervisor that is to be duplicated.

newfd This field determines which file descriptor number is allocated in the target. If the SECCOMP_ADDFD_FLAG_SETFD flag is set, then this field specifies which file de? scriptor number should be allocated. If this file descriptor number is already open in the target, it is atomically closed and reused. If the descriptor duplication fails due to an LSM check, or if srcfd is not a valid file descriptor, the file de? scriptor newfd will not be closed in the target process.

If the SECCOMP_ADDFD_FLAG_SETFD flag it not set, then this field must be 0, and the kernel allocates the lowest unused file descriptor number in the target.

newfd_flags

This field is a bit mask specifying flags that should be set on the file descriptor that is received in the target process. Currently, only the following flag is imple?

mented:

O_CLOEXEC

Set the close-on-exec flag on the received file descriptor.

On success, this ioctl(2) call returns the number of the file descriptor that was allocated in the target. Assuming that the emulated system call is one that returns a file descriptor as its function result (e.g., socket(2)), this value can be used as the return value (resp.val) that is supplied in the response that is subsequently sent with the SEC? COMP_IOCTL_NOTIF_SEND operation.

On error, -1 is returned and errno is set to indicate the error.

This operation can fail with the following errors:

EBADF Allocating the file descriptor in the target would cause the target's RLIMIT_NOFILE limit to be exceeded (see getrlimit(2)).

EBUSY If the flag SECCOMP_IOCTL_NOTIF_SEND is used, this means the operation can't proceed until other SECCOMP_IOCTL_NOTIF_ADDFD requests are processed.

EINPROGRESS

The user-space notification specified in the id field exists but has not yet been fetched (by a SECCOMP_IOCTL_NOTIF_RECV) or has already been responded to (by a SEC? COMP_IOCTL_NOTIF_SEND).

EINVAL An invalid flag was specified in the flags or newfd_flags field, or the newfd field is nonzero and the SECCOMP_ADDFD_FLAG_SETFD flag was not specified in the flags field.

EMFILE The file descriptor number specified in newfd exceeds the limit specified in /proc/sys/fs/nr_open.

ENOENT The blocked system call in the target has been interrupted by a signal handler or the target has terminated.

Here is some sample code (with error handling omitted) that uses the SEC? COMP_ADDFD_FLAG_SETFD operation (here, to emulate a call to openat(2)):

```
int fd, removeFd;
fd = openat(req->data.args[0], path, req->data.args[2],
            req->data.args[3]);
struct seccomp_notif_addfd addfd;
addfd.id = req->id;  /* Cookie from SECCOMP_IOCTL_NOTIF_RECV */
addfd.srcfd = fd;
```

```
        addfd.newfd = 0;
        addfd.flags = 0;
        addfd.newfd_flags = O_CLOEXEC;
        targetFd = ioctl(notifyFd, SECCOMP_IOCTL_NOTIF_ADDFD, &addfd);
        close(fd);          /* No longer needed in supervisor */
        struct seccomp_notif_resp *resp;
           /* Code to allocate 'resp' omitted */
        resp->id = req->id;
        resp->error = 0;        /* "Success" */
        resp->val = targetFd;
        resp->flags = 0;
        ioctl(notifyFd, SECCOMP_IOCTL_NOTIF_SEND, resp);
```

NOTES

One example use case for the user-space notification mechanism is to allow a container man? ager (a process which is typically running with more privilege than the processes inside the container) to mount block devices or create device nodes for the container. The mount use case provides an example of where the SECCOMP_USER_NOTIF_FLAG_CONTINUE ioctl(2) operation is useful. Upon receiving a notification for the mount(2) system call, the container manager (the "supervisor") can distinguish a request to mount a block filesystem (which would not be possible for a "target" process inside the container) and mount that file system. If, on the other hand, the container manager detects that the operation could be performed by the process inside the container (e.g., a mount of a tmpfs(5) filesystem), it can notify the kernel that the target process's mount(2) system call can continue.

select()/poll()/epoll semantics

The file descriptor returned when seccomp(2) is employed with the SECCOMP_FIL? TER_FLAG_NEW_LISTENER flag can be monitored using poll(2), epoll(7), and select(2). These interfaces indicate that the file descriptor is ready as follows:

? When a notification is pending, these interfaces indicate that the file descriptor is readable. Following such an indication, a subsequent SECCOMP_IOCTL_NOTIF_RECV ioctl(2) will not block, returning either information about a notification or else failing with the error EINTR if the target has been killed by a signal or its system call has been in? terrupted by a signal handler.

? After the notification has been received (i.e., by the SECCOMP_IOCTL_NOTIF_RECV ioctl(2)

operation), these interfaces indicate that the file descriptor is writable, meaning that a notification response can be sent using the SECCOMP_IOCTL_NOTIF_SEND ioctl(2) opera? tion.

? After the last thread using the filter has terminated and been reaped using waitpid(2) (or similar), the file descriptor indicates an end-of-file condition (readable in se? lect(2); POLLHUP/EPOLLHUP in poll(2)/ epoll_wait(2)).

Design goals; use of SECCOMP_USER_NOTIF_FLAG_CONTINUE

The intent of the user-space notification feature is to allow system calls to be performed on behalf of the target. The target's system call should either be handled by the supervi? sor or allowed to continue normally in the kernel (where standard security policies will be applied).

Note well: this mechanism must not be used to make security policy decisions about the sys? tem call, which would be inherently race-prone for reasons described next.

The SECCOMP_USER_NOTIF_FLAG_CONTINUE flag must be used with caution. If set by the supervi? sor, the target's system call will continue. However, there is a time-of-check, time-of-use race here, since an attacker could exploit the interval of time where the target is blocked waiting on the "continue" response to do things such as rewriting the system call arguments.

Note furthermore that a user-space notifier can be bypassed if the existing filters allow the use of seccomp(2) or prctl(2) to install a filter that returns an action value with a higher precedence than SECCOMP_RET_USER_NOTIF (see seccomp(2)).

It should thus be absolutely clear that the seccomp user-space notification mechanism can not be used to implement a security policy! It should only ever be used in scenarios where a more privileged process supervises the system calls of a lesser privileged target to get around kernel-enforced security restrictions when the supervisor deems this safe. In other words, in order to continue a system call, the supervisor should be sure that another secu? rity mechanism or the kernel itself will sufficiently block the system call if its arguments are rewritten to something unsafe.

Caveats regarding the use of /proc/tid/mem The discussion above noted the need to use the SECCOMP_IOCTL_NOTIF_ID_VALID ioctl(2) when opening the /proc/tid/mem file of the target to avoid the possibility of ac? cessing the memory of the wrong process in the event that the target terminates and its ID is recycled by another (unrelated) thread. However, the use of this ioctl(2) operation is also necessary in other situations, as explained in the following paragraphs.

Consider the following scenario, where the supervisor tries to read the pathname argument of a target's blocked mount(2) system call:

(1) From one of its functions (func()), the target calls mount(2), which triggers a user-space notification and causes the target to block.

(2) The supervisor receives the notification, opens /proc/tid/mem, and (successfully) per?forms the SECCOMP_IOCTL_NOTIF_ID_VALID check.

(3) The target receives a signal, which causes the mount(2) to abort.

(4) The signal handler executes in the target, and returns.

(5) Upon return from the handler, the execution of func() resumes, and it returns (and per?haps other functions are called, overwriting the memory that had been used for the stack frame of func()).

(6) Using the address provided in the notification information, the supervisor reads from the target's memory location that used to contain the pathname.

(7) The supervisor now calls mount(2) with some arbitrary bytes obtained in the previous step.

The conclusion from the above scenario is this: since the target's blocked system call may be interrupted by a signal handler, the supervisor must be written to expect that the target may abandon its system call at any time; in such an event, any information that the supervi?sor obtained from the target's memory must be considered invalid.

To prevent such scenarios, every read from the target's memory must be separated from use of the bytes so obtained by a SECCOMP_IOCTL_NOTIF_ID_VALID check. In the above example, the check would be placed between the two final steps. An example of such a check is shown in EXAMPLES.

Following on from the above, it should be clear that a write by the supervisor into the tar?get's memory can never be considered safe.

Caveats regarding blocking system calls

Suppose that the target performs a blocking system call (e.g., accept(2)) that the supervi?sor should handle. The supervisor might then in turn execute the same blocking system call. In this scenario, it is important to note that if the target's system call is now inter?rupted by a signal, the supervisor is not informed of this. If the supervisor does not take suitable steps to actively discover that the target's system call has been canceled, various difficulties can occur. Taking the example of accept(2), the supervisor might remain blocked in its accept(2) holding a port number that the target (which, after the interrup?

tion by the signal handler, perhaps closed its listening socket) might expect to be able to reuse in a bind(2) call.

Therefore, when the supervisor wishes to emulate a blocking system call, it must do so in such a way that it gets informed if the target's system call is interrupted by a signal han? dler. For example, if the supervisor itself executes the same blocking system call, then it could employ a separate thread that uses the SECCOMP_IOCTL_NOTIF_ID_VALID operation to check if the target is still blocked in its system call. Alternatively, in the accept(2) example, the supervisor might use poll(2) to monitor both the notification file descriptor (so as to discover when the target's accept(2) call has been interrupted) and the listening file de? scriptor (so as to know when a connection is available).

If the target's system call is interrupted, the supervisor must take care to release re? sources (e.g., file descriptors) that it acquired on behalf of the target.

Interaction with SA_RESTART signal handlers

Consider the following scenario:

(1) The target process has used sigaction(2) to install a signal handler with the SA_RESTART flag.

(2) The target has made a system call that triggered a seccomp user-space notification and the target is currently blocked until the supervisor sends a notification response.

(3) A signal is delivered to the target and the signal handler is executed.

(4) When (if) the supervisor attempts to send a notification response, the SEC? COMP_IOCTL_NOTIF_SEND ioctl(2)) operation will fail with the ENOENT error.

In this scenario, the kernel will restart the target's system call. Consequently, the su? pervisor will receive another user-space notification. Thus, depending on how many times the blocked system call is interrupted by a signal handler, the supervisor may receive mul? tiple notifications for the same instance of a system call in the target.

One oddity is that system call restarting as described in this scenario will occur even for the blocking system calls listed in signal(7) that would never normally be restarted by the SA_RESTART flag.

Furthermore, if the supervisor response is a file descriptor added with SECCOMP_IOCTL_NO? TIF_ADDFD, then the flag SECCOMP_ADDFD_FLAG_SEND can be used to atomically add the file de? scriptor and return that value, making sure no file descriptors are inadvertently leaked into the target.

If a SECCOMP_IOCTL_NOTIF_RECV ioctl(2) operation is performed after the target terminates, then the ioctl(2) call simply blocks (rather than returning an error to indicate that the target no longer exists).

EXAMPLES

The (somewhat contrived) program shown below demonstrates the use of the interfaces described in this page. The program creates a child process that serves as the "target" process. The child process installs a seccomp filter that returns the SECCOMP_RET_USER_NOTIF action value if a call is made to mkdir(2). The child process then calls mkdir(2) once for each of the supplied command-line arguments, and reports the result returned by the call. After processing all arguments, the child process terminates.

The parent process acts as the supervisor, listening for the notifications that are generated when the target process calls mkdir(2). When such a notification occurs, the supervisor examines the memory of the target process (using /proc/pid/mem) to discover the pathname argument that was supplied to the mkdir(2) call, and performs one of the following actions:

- If the pathname begins with the prefix "/tmp/", then the supervisor attempts to create the specified directory, and then spoofs a return for the target process based on the return value of the supervisor's mkdir(2) call. In the event that that call succeeds, the spoofed success return value is the length of the pathname.
- If the pathname begins with "./" (i.e., it is a relative pathname), the supervisor sends a SECCOMP_USER_NOTIF_FLAG_CONTINUE response to the kernel to say that the kernel should execute the target process's mkdir(2) call.
- If the pathname begins with some other prefix, the supervisor spoofs an error return for the target process, so that the target process's mkdir(2) call appears to fail with the error EOPNOTSUPP ("Operation not supported"). Additionally, if the specified pathname is exactly "/bye", then the supervisor terminates.

This program can be used to demonstrate various aspects of the behavior of the seccomp user-space notification mechanism. To help aid such demonstrations, the program logs various messages to show the operation of the target process (lines prefixed "T:") and the supervisor (indented lines prefixed "S:").

In the following example, the target attempts to create the directory /tmp/x. Upon receiving the notification, the supervisor creates the directory on the target's behalf, and spoofs a success return to be received by the target process's mkdir(2) call.

```
$ ./seccomp_unotify /tmp/x;
```

T: PID = 23168

T: about to mkdir("/tmp/x")

S: got notification (ID 0x17445c4a0f4e0e3c) for PID 23168

S: executing: mkdir("/tmp/x", 0700)

S: success! spoofed return = 6

S: sending response (flags = 0; val = 6; error = 0)

T: SUCCESS: mkdir(2) returned 6

T: terminating

S: target has terminated; bye

In the above output, note that the spoofed return value seen by the target process is 6 (the length of the pathname /tmp/x), whereas a normal mkdir(2) call returns 0 on success.

In the next example, the target attempts to create a directory using the relative pathname ./sub. Since this pathname starts with "./", the supervisor sends a SECCOMP_USER_NO? TIF_FLAG_CONTINUE response to the kernel, and the kernel then (successfully) executes the target process's mkdir(2) call.

$ ./seccomp_unotify ./sub;

T: PID = 23204

T: about to mkdir("./sub")

S: got notification (ID 0xddb16abe25b4c12) for PID 23204

S: target can execute system call

S: sending response (flags = 0x1; val = 0; error = 0)

T: SUCCESS: mkdir(2) returned 0

T: terminating

S: target has terminated; bye

If the target process attempts to create a directory with a pathname that doesn't start with "." and doesn't begin with the prefix "/tmp/", then the supervisor spoofs an error return (EOPNOTSUPP, "Operation not supported") for the target's mkdir(2) call (which is not exe? cuted):

$ ./seccomp_unotify /xxx;

T: PID = 23178

T: about to mkdir("/xxx")

S: got notification (ID 0xe7dc095d1c524e80) for PID 23178

S: spoofing error response (Operation not supported)

S: sending response (flags = 0; val = 0; error = -95)

T: ERROR: mkdir(2): Operation not supported

T: terminating

S: target has terminated; bye

In the next example, the target process attempts to create a directory with the pathname /tmp/nosuchdir/b. Upon receiving the notification, the supervisor attempts to create that directory, but the mkdir(2) call fails because the directory /tmp/nosuchdir does not exist. Consequently, the supervisor spoofs an error return that passes the error that it received back to the target process's mkdir(2) call.

$ ./seccomp_unotify /tmp/nosuchdir/b;

T: PID = 23199

T: about to mkdir("/tmp/nosuchdir/b")

S: got notification (ID 0x8744454293506046) for PID 23199

S: executing: mkdir("/tmp/nosuchdir/b", 0700)

S: failure! (errno = 2; No such file or directory)

S: sending response (flags = 0; val = 0; error = -2)

T: ERROR: mkdir(2): No such file or directory

T: terminating

S: target has terminated; bye

If the supervisor receives a notification and sees that the argument of the target's mkdir(2) is the string "/bye", then (as well as spoofing an EOPNOTSUPP error), the supervi? sor terminates. If the target process subsequently executes another mkdir(2) that triggers its seccomp filter to return the SECCOMP_RET_USER_NOTIF action value, then the kernel causes the target process's system call to fail with the error ENOSYS ("Function not implemented"). This is demonstrated by the following example:

$ ./seccomp_unotify /bye /tmp/y;

T: PID = 23185

T: about to mkdir("/bye")

S: got notification (ID 0xa81236b1d2f7b0f4) for PID 23185

S: spoofing error response (Operation not supported)

S: sending response (flags = 0; val = 0; error = -95)

S: terminating **********

T: ERROR: mkdir(2): Operation not supported

```
        T: about to mkdir("/tmp/y")
        T: ERROR: mkdir(2): Function not implemented
        T: terminating
```

Program source

```
    #define _GNU_SOURCE
    #include <err.h>
    #include <errno.h>
    #include <fcntl.h>
    #include <limits.h>
    #include <linux/audit.h>
    #include <linux/filter.h>
    #include <linux/seccomp.h>
    #include <signal.h>
    #include <stdbool.h>
    #include <stdcountof.h>
    #include <stddef.h>
    #include <stdint.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <sys/ioctl.h>
    #include <sys/prctl.h>
    #include <sys/socket.h>
    #include <sys/stat.h>
    #include <sys/syscall.h>
    #include <sys/types.h>
    #include <sys/un.h>
    #include <unistd.h>
    /* Send the file descriptor 'fd' over the connected UNIX domain socket
       'sockfd'.  Returns 0 on success, or -1 on error.  */
    static int
    sendfd(int sockfd, int fd)
    {
```

```
int             data;
struct iovec    iov;
struct msghdr   msgh;
struct cmsghdr  *cmsgp;
/* Allocate a char array of suitable size to hold the ancillary data.
   However, since this buffer is in reality a 'struct cmsghdr', use a
   union to ensure that it is suitably aligned.  */
union {
    char   buf[CMSG_SPACE(sizeof(int))];
                /* Space large enough to hold an 'int' */
    struct cmsghdr align;
} controlMsg;
/* The 'msg_name' field can be used to specify the address of the
   destination socket when sending a datagram.  However, we do not
   need to use this field because 'sockfd' is a connected socket.  */
msgh.msg_name = NULL;
msgh.msg_namelen = 0;
/* On Linux, we must transmit at least one byte of real data in
   order to send ancillary data.  We transmit an arbitrary integer
   whose value is ignored by recvfd().  */
msgh.msg_iov = &iov;
msgh.msg_iovlen = 1;
iov.iov_base = &data;
iov.iov_len = sizeof(int);
data = 12345;
/* Set 'msghdr' fields that describe ancillary data */
msgh.msg_control = controlMsg.buf;
msgh.msg_controllen = sizeof(controlMsg.buf);
/* Set up ancillary data describing file descriptor to send */
cmsgp = CMSG_FIRSTHDR(&msgh);
cmsgp->cmsg_level = SOL_SOCKET;
cmsgp->cmsg_type = SCM_RIGHTS;
cmsgp->cmsg_len = CMSG_LEN(sizeof(int));
```

```
    memcpy(CMSG_DATA(cmsgp), &fd, sizeof(int));
    /* Send real plus ancillary data */
    if (sendmsg(sockfd, &msgh, 0) == -1)
        return -1;
    return 0;
}
/* Receive a file descriptor on a connected UNIX domain socket.  Returns
   the received file descriptor on success, or -1 on error.  */
static int
recvfd(int sockfd)
{
    int            data, fd;
    ssize_t        nr;
    struct iovec   iov;
    struct msghdr  msgh;
    /* Allocate a char buffer for the ancillary data.  See the comments
       in sendfd() */
    union {
        char   buf[CMSG_SPACE(sizeof(int))];
        struct cmsghdr align;
    } controlMsg;
    struct cmsghdr *cmsgp;
    /* The 'msg_name' field can be used to obtain the address of the
       sending socket.  However, we do not need this information.  */
    msgh.msg_name = NULL;
    msgh.msg_namelen = 0;
    /* Specify buffer for receiving real data */
    msgh.msg_iov = &iov;
    msgh.msg_iovlen = 1;
    iov.iov_base = &data;       /* Real data is an 'int' */
    iov.iov_len = sizeof(int);
    /* Set 'msghdr' fields that describe ancillary data */
    msgh.msg_control = controlMsg.buf;
```

```
    msgh.msg_controllen = sizeof(controlMsg.buf);
    /* Receive real plus ancillary data; real data is ignored */
    nr = recvmsg(sockfd, &msgh, 0);
    if (nr == -1)
        return -1;
    cmsgp = CMSG_FIRSTHDR(&msgh);
    /* Check the validity of the 'cmsghdr' */
    if (cmsgp == NULL
        || cmsgp->cmsg_len != CMSG_LEN(sizeof(int))
        || cmsgp->cmsg_level != SOL_SOCKET
        || cmsgp->cmsg_type != SCM_RIGHTS)
    {
        errno = EINVAL;
        return -1;
    }
    /* Return the received file descriptor to our caller */
    memcpy(&fd, CMSG_DATA(cmsgp), sizeof(int));
    return fd;
}
static void
sigchldHandler(int sig)
{
    char msg[] = "\tS: target has terminated; bye\n";
    write(STDOUT_FILENO, msg, sizeof(msg) - 1);
    _exit(EXIT_SUCCESS);
}
static int
seccomp(unsigned int operation, unsigned int flags, void *args)
{
    return syscall(SYS_seccomp, operation, flags, args);
}
/* The following is the x86-64-specific BPF boilerplate code for checking
   that the BPF program is running on the right architecture + ABI.  At
```

```
   completion of these instructions, the accumulator contains the system
   call number.  */
/* For the x32 ABI, all system call numbers have bit 30 set */
#define X32_SYSCALL_BIT         0x40000000
#define X86_64_CHECK_ARCH_AND_LOAD_SYSCALL_NR \
        BPF_STMT(BPF_LD | BPF_W | BPF_ABS, \
                (offsetof(struct seccomp_data, arch))), \
        BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, AUDIT_ARCH_X86_64, 0, 2), \
        BPF_STMT(BPF_LD | BPF_W | BPF_ABS, \
                (offsetof(struct seccomp_data, nr))), \
        BPF_JUMP(BPF_JMP | BPF_JGE | BPF_K, X32_SYSCALL_BIT, 0, 1), \
        BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL_PROCESS)
/* installNotifyFilter() installs a seccomp filter that generates
   user-space notifications (SECCOMP_RET_USER_NOTIF) when the process
   calls mkdir(2); the filter allows all other system calls.
   The function return value is a file descriptor from which the
   user-space notifications can be fetched.  */
static int
installNotifyFilter(void)
{
    int notifyFd;
    struct sock_filter filter[] = {
        X86_64_CHECK_ARCH_AND_LOAD_SYSCALL_NR,
        /* mkdir() triggers notification to user-space supervisor */
        BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, SYS_mkdir, 0, 1),
        BPF_STMT(BPF_RET + BPF_K, SECCOMP_RET_USER_NOTIF),
        /* Every other system call is allowed */
        BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
    };
    struct sock_fprog prog = {
        .len = countof(filter),
        .filter = filter,
    };
```

```
    /* Install the filter with the SECCOMP_FILTER_FLAG_NEW_LISTENER flag;
       as a result, seccomp() returns a notification file descriptor.  */
    notifyFd = seccomp(SECCOMP_SET_MODE_FILTER,
                       SECCOMP_FILTER_FLAG_NEW_LISTENER, &prog);
    if (notifyFd == -1)
        err(EXIT_FAILURE, "seccomp-install-notify-filter");
    return notifyFd;
}
/* Close a pair of sockets created by socketpair() */
static void
closeSocketPair(int sockPair[2])
{
    if (close(sockPair[0]) == -1)
        err(EXIT_FAILURE, "closeSocketPair-close-0");
    if (close(sockPair[1]) == -1)
        err(EXIT_FAILURE, "closeSocketPair-close-1");
}
/* Implementation of the target process.  Create a child process that:
   (1) installs a seccomp filter with the
       SECCOMP_FILTER_FLAG_NEW_LISTENER flag;
   (2) writes the seccomp notification file descriptor returned from
       the previous step onto the UNIX domain socket, 'sockPair[0]';
   (3) calls mkdir(2) for each element of 'argv'.
   The function return value in the parent is the PID of the child
   process; the child does not return from this function.  */
static pid_t
targetProcess(int sockPair[2], char *argv[])
{
    int    notifyFd, s;
    pid_t  targetPid;
    targetPid = fork();
    if (targetPid == -1)
        err(EXIT_FAILURE, "fork");
```

```
    if (targetPid > 0)          /* In parent, return PID of child */
        return targetPid;
    /* Child falls through to here */
    printf("T: PID = %ld\n", (long) getpid());
    /* Install seccomp filter(s) */
    if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0))
        err(EXIT_FAILURE, "prctl");
    notifyFd = installNotifyFilter();
    /* Pass the notification file descriptor to the tracing process over
       a UNIX domain socket */
    if (sendfd(sockPair[0], notifyFd) == -1)
        err(EXIT_FAILURE, "sendfd");
    /* Notification and socket FDs are no longer needed in target */
    if (close(notifyFd) == -1)
        err(EXIT_FAILURE, "close-target-notify-fd");
    closeSocketPair(sockPair);
    /* Perform a mkdir() call for each of the command-line arguments */
    for (char **ap = argv; *ap != NULL; ap++) {
        printf("\nT: about to mkdir(\"%s\")\n", *ap);
        s = mkdir(*ap, 0700);
        if (s == -1)
            perror("T: ERROR: mkdir(2)");
        else
            printf("T: SUCCESS: mkdir(2) returned %d\n", s);
    }
    printf("\nT: terminating\n");
    exit(EXIT_SUCCESS);
}
/* Check that the notification ID provided by a SECCOMP_IOCTL_NOTIF_RECV
   operation is still valid.  It will no longer be valid if the target
   process has terminated or is no longer blocked in the system call that
   generated the notification (because it was interrupted by a signal).
   This operation can be used when doing such things as accessing
```

```
   /proc/PID files in the target process in order to avoid TOCTOU race
   conditions where the PID that is returned by SECCOMP_IOCTL_NOTIF_RECV
   terminates and is reused by another process.  */
static bool
cookieIsValid(int notifyFd, uint64_t id)
{
    return ioctl(notifyFd, SECCOMP_IOCTL_NOTIF_ID_VALID, &id) == 0;
}
/* Access the memory of the target process in order to fetch the
   pathname referred to by the system call argument 'argNum' in
   'req->data.args[]'.  The pathname is returned in 'path',
   a buffer of 'size' bytes allocated by the caller.
   Returns true if the pathname is successfully fetched, and false
   otherwise.  For possible causes of failure, see the comments below.  */
static bool
getTargetPathname(struct seccomp_notif *req, int notifyFd,
                  int argNum, char *path, size_t size)
{
    int      procMemFd;
    char     procMemPath[PATH_MAX];
    ssize_t  nread;
    snprintf(procMemPath, sizeof(procMemPath), "/proc/%d/mem", req->pid);
    procMemFd = open(procMemPath, O_RDONLY | O_CLOEXEC);
    if (procMemFd == -1)
        return false;
    /* Check that the process whose info we are accessing is still alive
       and blocked in the system call that caused the notification.
       If the SECCOMP_IOCTL_NOTIF_ID_VALID operation (performed in
       cookieIsValid()) succeeded, we know that the /proc/PID/mem file
       descriptor that we opened corresponded to the process for which we
       received a notification.  If that process subsequently terminates,
       then read() on that file descriptor will return 0 (EOF).  */
    if (!cookieIsValid(notifyFd, req->id)) {
```

```
        close(procMemFd);
        return false;
    }
    /* Read bytes at the location containing the pathname argument */
    nread = pread(procMemFd, path, size, req->data.args[argNum]);
    close(procMemFd);
    if (nread <= 0)
        return false;
    /* Once again check that the notification ID is still valid.  The
       case we are particularly concerned about here is that just
       before we fetched the pathname, the target's blocked system
       call was interrupted by a signal handler, and after the handler
       returned, the target carried on execution (past the interrupted
       system call).  In that case, we have no guarantees about what we
       are reading, since the target's memory may have been arbitrarily
       changed by subsequent operations.  */
    if (!cookieIsValid(notifyFd, req->id)) {
        perror("\tS: notification ID check failed!!!");
        return false;
    }
    /* Even if the target's system call was not interrupted by a signal,
       we have no guarantees about what was in the memory of the target
       process.  (The memory may have been modified by another thread, or
       even by an external attacking process.) We therefore treat the
       buffer returned by pread() as untrusted input.  The buffer should
       contain a terminating null byte; if not, then we will trigger an
       error for the target process.  */
    if (strnlen(path, nread) < nread)
        return true;
    return false;
}
/* Allocate buffers for the seccomp user-space notification request and
   response structures.  It is the caller's responsibility to free the
```

```
   buffers returned via 'req' and 'resp'.  */
static void
allocSeccompNotifBuffers(struct seccomp_notif **req,
                         struct seccomp_notif_resp **resp,
                         struct seccomp_notif_sizes *sizes)
{
    size_t  resp_size;
    /* Discover the sizes of the structures that are used to receive
       notifications and send notification responses, and allocate
       buffers of those sizes.  */
    if (seccomp(SECCOMP_GET_NOTIF_SIZES, 0, sizes) == -1)
        err(EXIT_FAILURE, "seccomp-SECCOMP_GET_NOTIF_SIZES");
    *req = malloc(sizes->seccomp_notif);
    if (*req == NULL)
        err(EXIT_FAILURE, "malloc-seccomp_notif");
    /* When allocating the response buffer, we must allow for the fact
       that the user-space binary may have been built with user-space
       headers where 'struct seccomp_notif_resp' is bigger than the
       response buffer expected by the (older) kernel.  Therefore, we
       allocate a buffer that is the maximum of the two sizes.  This
       ensures that if the supervisor places bytes into the response
       structure that are past the response size that the kernel expects,
       then the supervisor is not touching an invalid memory location.  */
    resp_size = sizes->seccomp_notif_resp;
    if (sizeof(struct seccomp_notif_resp) > resp_size)
        resp_size = sizeof(struct seccomp_notif_resp);
    *resp = malloc(resp_size);
    if (*resp == NULL)
        err(EXIT_FAILURE, "malloc-seccomp_notif_resp");
}
/* Handle notifications that arrive via the SECCOMP_RET_USER_NOTIF file
   descriptor, 'notifyFd'.  */
static void
```

```
handleNotifications(int notifyFd)
{
    bool                    pathOK;
    char                    path[PATH_MAX];
    struct seccomp_notif        *req;
    struct seccomp_notif_resp   *resp;
    struct seccomp_notif_sizes  sizes;
    allocSeccompNotifBuffers(&req, &resp, &sizes);
    /* Loop handling notifications */
    for (;;) {
        /* Wait for next notification, returning info in '*req' */
        memset(req, 0, sizes.seccomp_notif);
        if (ioctl(notifyFd, SECCOMP_IOCTL_NOTIF_RECV, req) == -1) {
            if (errno == EINTR)
                continue;
            err(EXIT_FAILURE, "\tS: ioctl-SECCOMP_IOCTL_NOTIF_RECV");
        }
        printf("\tS: got notification (ID %#llx) for PID %d\n",
                req->id, req->pid);
        /* The only system call that can generate a notification event
           is mkdir(2).  Nevertheless, we check that the notified system
           call is indeed mkdir() as kind of future-proofing of this
           code in case the seccomp filter is later modified to
           generate notifications for other system calls.  */
        if (req->data.nr != SYS_mkdir) {
            printf("\tS: notification contained unexpected "
                    "system call number; bye!!!\n");
            exit(EXIT_FAILURE);
        }
        pathOK = getTargetPathname(req, notifyFd, 0, path, sizeof(path));
        /* Prepopulate some fields of the response */
        resp->id = req->id;     /* Response includes notification ID */
        resp->flags = 0;
```

```
    resp->val = 0;
    /* If getTargetPathname() failed, trigger an EINVAL error
       response (sending this response may yield an error if the
       failure occurred because the notification ID was no longer
       valid); if the directory is in /tmp, then create it on behalf
       of the supervisor; if the pathname starts with '.', tell the
       kernel to let the target process execute the mkdir();
       otherwise, give an error for a directory pathname in any other
       location.  */
    if (!pathOK) {
        resp->error = -EINVAL;
        printf("\tS: spoofing error for invalid pathname (%s)\n",
                strerror(-resp->error));
    } else if (strncmp(path, "/tmp/", strlen("/tmp/")) == 0) {
        printf("\tS: executing: mkdir(\"%s\", %#llo)\n",
                path, req->data.args[1]);
        if (mkdir(path, req->data.args[1]) == 0) {
            resp->error = 0;            /* "Success" */
            resp->val = strlen(path);   /* Used as return value of
                                           mkdir() in target */
            printf("\tS: success! spoofed return = %lld\n",
                    resp->val);
        } else {
            /* If mkdir() failed in the supervisor, pass the error
               back to the target */
            resp->error = -errno;
            printf("\tS: failure! (errno = %d; %s)\n", errno,
                    strerror(errno));
        }
    } else if (strncmp(path, "./", strlen("./")) == 0) {
        resp->error = resp->val = 0;
        resp->flags = SECCOMP_USER_NOTIF_FLAG_CONTINUE;
        printf("\tS: target can execute system call\n");
```

```
        } else {
            resp->error = -EOPNOTSUPP;
            printf("\tS: spoofing error response (%s)\n",
                    strerror(-resp->error));
        }
        /* Send a response to the notification */
        printf("\tS: sending response "
               "(flags = %#x; val = %lld; error = %d)\n",
               resp->flags, resp->val, resp->error);
        if (ioctl(notifyFd, SECCOMP_IOCTL_NOTIF_SEND, resp) == -1) {
            if (errno == ENOENT)
                printf("\tS: response failed with ENOENT; "
                       "perhaps target process's syscall was "
                       "interrupted by a signal?\n");
            else
                perror("ioctl-SECCOMP_IOCTL_NOTIF_SEND");
        }
        /* If the pathname is just "/bye", then the supervisor breaks out
           of the loop and terminates.  This allows us to see what happens
           if the target process makes further calls to mkdir(2).  */
        if (strcmp(path, "/bye") == 0)
            break;
    }
    free(req);
    free(resp);
    printf("\tS: terminating **********\n");
    exit(EXIT_FAILURE);
}
/* Implementation of the supervisor process:
   (1) obtains the notification file descriptor from 'sockPair[1]'
   (2) handles notifications that arrive on that file descriptor.  */
static void
supervisor(int sockPair[2])
```

```
{
    int notifyFd;
    notifyFd = recvfd(sockPair[1]);
    if (notifyFd == -1)
        err(EXIT_FAILURE, "recvfd");
    closeSocketPair(sockPair);  /* We no longer need the socket pair */
    handleNotifications(notifyFd);
}
int
main(int argc, char *argv[])
{
    int              sockPair[2];
    struct sigaction  sa;
    setbuf(stdout, NULL);
    if (argc < 2) {
        fprintf(stderr, "At least one pathname argument is required\n");
        exit(EXIT_FAILURE);
    }
    /* Create a UNIX domain socket that is used to pass the seccomp
       notification file descriptor from the target process to the
       supervisor process.  */
    if (socketpair(AF_UNIX, SOCK_STREAM, 0, sockPair) == -1)
        err(EXIT_FAILURE, "socketpair");
    /* Create a child process--the "target"--that installs seccomp
       filtering.  The target process writes the seccomp notification
       file descriptor onto 'sockPair[0]' and then calls mkdir(2) for
       each directory in the command-line arguments.  */
    (void) targetProcess(sockPair, &argv[optind]);
    /* Catch SIGCHLD when the target terminates, so that the
       supervisor can also terminate.  */
    sa.sa_handler = sigchldHandler;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
```

```
        if (sigaction(SIGCHLD, &sa, NULL) == -1)
            err(EXIT_FAILURE, "sigaction");
        supervisor(sockPair);
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

ioctl(2), pidfd_getfd(2), pidfd_open(2), seccomp(2)

A further example program can be found in the kernel source file samples/seccomp/user-trap.c.