



Linux Ubuntu 26.04 Manual Pages on command 'openat2.2'

\$ man openat2.2

openat2(2) System Calls Manual openat2(2)

NAME

openat2 - open and possibly create a file (extended)

LIBRARY

Standard C library (libc, -lc)

SYNOPSIS

```
#include <fcntl.h>            /* Definition of O_* and S_* constants */
#include <linux/openat2.h> /* Definition of RESOLVE_* constants */
#include <sys/syscall.h>    /* Definition of SYS_* constants */
#include <unistd.h>

long syscall(SYS_openat2, int dirfd, const char *path,
             struct open_how *how, size_t size);
```

Note: glibc provides no wrapper for openat2(), necessitating the use of syscall(2).

DESCRIPTION

The openat2() system call is an extension of openat(2) and provides a superset of its functionality.

The openat2() system call opens the file specified by path. If the specified file does not exist, it may optionally (if O_CREAT is specified in how.flags) be created.

As with openat(2), if path is relative, then it is interpreted relative to the directory referred to by the file descriptor dirfd (or the current working directory of the calling process, if dirfd is the special value AT_FDCWD). If path is absolute, then dirfd is ignored (unless how.resolve contains RESOLVE_IN_ROOT, in which case path is resolved relative to dirfd).

Rather than taking a single flags argument, an extensible structure (how) is passed to allow for future extensions. The size argument must be specified as `sizeof(struct open_how)`.

The `open_how` structure

The `how` argument specifies how path should be opened, and acts as a superset of the `flags` and `mode` arguments to `openat(2)`. This argument is a pointer to an `open_how` structure, described in `open_how(2)type`.

Any future extensions to `openat2()` will be implemented as new fields appended to the `open_how` structure, with a zero value in a new field resulting in the kernel behaving as though that extension field was not present. Therefore, the caller must zero-fill this structure on initialization. (See the "Extensibility" section of the NOTES for more detail on why this is necessary.)

The fields of the `open_how` structure are as follows:

flags This field specifies the file creation and file status flags to use when opening the file. All of the `O_*` flags defined for `openat(2)` are valid `openat2()` flag values.

Whereas `openat(2)` ignores unknown bits in its flags argument, `openat2()` returns an error if unknown or conflicting flags are specified in `how.flags`.

mode This field specifies the mode for the new file, with identical semantics to the `mode` argument of `openat(2)`.

Whereas `openat(2)` ignores bits other than those in the range `07777` in its mode argument, `openat2()` returns an error if `how.mode` contains bits other than `07777`. Similarly, an error is returned if `openat2()` is called with a nonzero `how.mode` and `how.flags` does not contain `O_CREAT` or `O_TMPFILE`.

resolve

This is a bit-mask of flags that modify the way in which all components of path will be resolved. (See `path_resolution(7)` for background information.)

The primary use case for these flags is to allow trusted programs to restrict how untrusted paths (or paths inside untrusted directories) are resolved. The full list of resolve flags is as follows:

RESOLVE_BENEATH

Do not permit the path resolution to succeed if any component of the resolution is not a descendant of the directory indicated by `dirfd`. This causes absolute symbolic links (and absolute values of path) to be rejected.

Currently, this flag also disables magic-link resolution (see below). How?

ever, this may change in the future. Therefore, to ensure that magic links are not resolved, the caller should explicitly specify `RESOLVE_NO_MAGICLINKS`.

`RESOLVE_IN_ROOT`

Treat the directory referred to by `dirfd` as the root directory while resolving path. Absolute symbolic links are interpreted relative to `dirfd`. If a prefix component of path equates to `dirfd`, then an immediately following `..` component likewise equates to `dirfd` (just as `./` is traditionally equivalent to `/`). If path is absolute, it is also interpreted relative to `dirfd`.

The effect of this flag is as though the calling process had used `chroot(2)` to (temporarily) modify its root directory (to the directory referred to by `dirfd`). However, unlike `chroot(2)` (which changes the filesystem root permanently for a process), `RESOLVE_IN_ROOT` allows a program to efficiently re-strict path resolution on a per-open basis.

Currently, this flag also disables magic-link resolution. However, this may change in the future. Therefore, to ensure that magic links are not resolved, the caller should explicitly specify `RESOLVE_NO_MAGICLINKS`.

`RESOLVE_NO_MAGICLINKS`

Disallow all magic-link resolution during path resolution.

Magic links are symbolic link-like objects that are most notably found in `/proc(5)`; examples include `/proc/pid/exe` and `/proc/pid/fd/*`. (See `symlink(7)` for more details.)

Unknowingly opening magic links can be risky for some applications. Examples of such risks include the following:

- ? If the process opening a pathname is a controlling process that currently has no controlling terminal (see `credentials(7)`), then opening a magic link inside `/proc/pid/fd` that happens to refer to a terminal would cause the process to acquire a controlling terminal.
- ? In a containerized environment, a magic link inside `/proc` may refer to an object outside the container, and thus may provide a means to escape from the container.

Because of such risks, an application may prefer to disable magic link resolution using the `RESOLVE_NO_MAGICLINKS` flag.

If the trailing component (i.e., `basename`) of path is a magic link, how.re?

solve contains RESOLVE_NO_MAGICLINKS, and how.flags contains both O_PATH and O_NOFOLLOW, then an O_PATH file descriptor referencing the magic link will be returned.

RESOLVE_NO_SYMLINKS

Disallow resolution of symbolic links during path resolution. This option implies RESOLVE_NO_MAGICLINKS.

If the trailing component (i.e., basename) of path is a symbolic link, how.resolve contains RESOLVE_NO_SYMLINKS, and how.flags contains both O_PATH and O_NOFOLLOW, then an O_PATH file descriptor referencing the symbolic link will be returned.

Note that the effect of the RESOLVE_NO_SYMLINKS flag, which affects the treatment of symbolic links in all of the components of path, differs from the effect of the O_NOFOLLOW file creation flag (in how.flags), which affects the handling of symbolic links only in the final component of path.

Applications that employ the RESOLVE_NO_SYMLINKS flag are encouraged to make its use configurable (unless it is used for a specific security purpose), as symbolic links are very widely used by end-users. Setting this flag indiscriminately?i.e., for purposes not specifically related to security?for all uses of openat2() may result in spurious errors on previously functional systems. This may occur if, for example, a system pathname that is used by an application is modified (e.g., in a new distribution release) so that a path name component (now) contains a symbolic link.

RESOLVE_NO_XDEV

Disallow traversal of mount points during path resolution (including all bind mounts). Consequently, path must either be on the same mount as the directory referred to by dirfd, or on the same mount as the current working directory if dirfd is specified as AT_FDCWD.

Applications that employ the RESOLVE_NO_XDEV flag are encouraged to make its use configurable (unless it is used for a specific security purpose), as bind mounts are widely used by end-users. Setting this flag indiscriminately?i.e., for purposes not specifically related to security?for all uses of openat2() may result in spurious errors on previously functional systems. This may occur if, for example, a system pathname that is used by an application is modified

fied (e.g., in a new distribution release) so that a pathname component (now) contains a bind mount.

RESOLVE_CACHED (since Linux 5.12)

Make the open operation fail unless all path components are already present in the kernel's lookup cache. If any kind of revalidation or I/O is needed to satisfy the lookup, `openat2()` fails with the error `EAGAIN`. This is useful in providing a fast-path open that can be performed without resorting to thread offload, or other mechanisms that an application might use to offload slower operations.

If any bits other than those listed above are set in `how.resolve`, an error is returned.

RETURN VALUE

On success, a new file descriptor is returned. On error, -1 is returned, and `errno` is set to indicate the error.

ERRORS

The set of errors returned by `openat2()` includes all of the errors returned by `openat(2)`, as well as the following additional errors:

E2BIG An extension that this kernel does not support was specified in `how`. (See the "Extensibility" section of NOTES for more detail on how extensions are handled.)

EAGAIN `how.resolve` contains either `RESOLVE_IN_ROOT` or `RESOLVE_BENEATH`, and the kernel could not ensure that a `".."` component didn't escape (due to a race condition or potential attack). The caller may choose to retry the `openat2()` call.

EAGAIN `RESOLVE_CACHED` was set, and the open operation cannot be performed using only cached information. The caller should retry without `RESOLVE_CACHED` set in `how.resolve`.

EINVAL An unknown flag or invalid value was specified in `how`.

EINVAL `mode` is nonzero, but `how.flags` does not contain `O_CREAT` or `O_TMPFILE`.

EINVAL `size` was smaller than any known version of `struct open_how`.

ELOOP `how.resolve` contains `RESOLVE_NO_SYMLINKS`, and one of the path components was a symbolic link (or magic link).

ELOOP `how.resolve` contains `RESOLVE_NO_MAGICLINKS`, and one of the path components was a magic link.

EXDEV `how.resolve` contains either `RESOLVE_IN_ROOT` or `RESOLVE_BENEATH`, and an escape from the root during path resolution was detected.

EXDEV how.resolve contains RESOLVE_NO_XDEV, and a path component crosses a mount point.

STANDARDS

Linux.

HISTORY

Linux 5.6.

The semantics of RESOLVE_BENEATH were modeled after FreeBSD's O_BENEATH, but avoided a well-known correctness bug in FreeBSD's implementation that rendered it effectively insecure.

Later, FreeBSD 13 introduced O_RESOLVE_BENEATH to replace the insecure O_BENEATH. FreeBSD's O_RESOLVE_BENEATH semantics are based on Linux's RESOLVE_BENEATH and the two are now functionally equivalent.

NOTES

Extensibility

In order to allow for future extensibility, openat2() requires the user-space application to specify the size of the open_how structure that it is passing. By providing this information, it is possible for openat2() to provide both forwards- and backwards-compatibility, with size acting as an implicit version number. (Because new extension fields will always be appended, the structure size will always increase.) This extensibility design is very similar to other system calls such as sched_setattr(2), perf_event_open(2), and clone3(2). If we let usize be the size of the structure as specified by the user-space application, and ksize be the size of the structure which the kernel supports, then there are three cases to consider:

? If ksize equals usize, then there is no version mismatch and how can be used verbatim.

? If ksize is larger than usize, then there are some extension fields that the kernel supports which the user-space application is unaware of. Because a zero value in any added extension field signifies a no-op, the kernel treats all of the extension fields not provided by the user-space application as having zero values. This provides backwards-compatibility.

? If ksize is smaller than usize, then there are some extension fields which the user-space application is aware of but which the kernel does not support. Because any extension field must have its zero values signify a no-op, the kernel can safely ignore the unsupported extension fields if they are all-zero. If any unsupported extension fields are nonzero, then -1 is returned and errno is set to E2BIG. This provides forwards-compatibility.

Because the definition of struct open_how may change in the future (with new fields being added when system headers are updated), user-space applications should zero-fill struct open_how to ensure that recompiling the program with new headers will not result in spurious errors at run time. The simplest way is to use a designated initializer:

```
struct open_how how = { .flags = O_RDWR,  
                        .resolve = RESOLVE_IN_ROOT };
```

or explicitly using memset(3) or similar:

```
struct open_how how;  
  
memset(&how, 0, sizeof(how));  
  
how.flags = O_RDWR;  
  
how.resolve = RESOLVE_IN_ROOT;
```

A user-space application that wishes to determine which extensions the running kernel supports can do so by conducting a binary search on size with a structure which has every byte nonzero (to find the largest value which doesn't produce an error of E2BIG).

SEE ALSO

openat(2), open_how(2type), path_resolution(7), symlink(7)