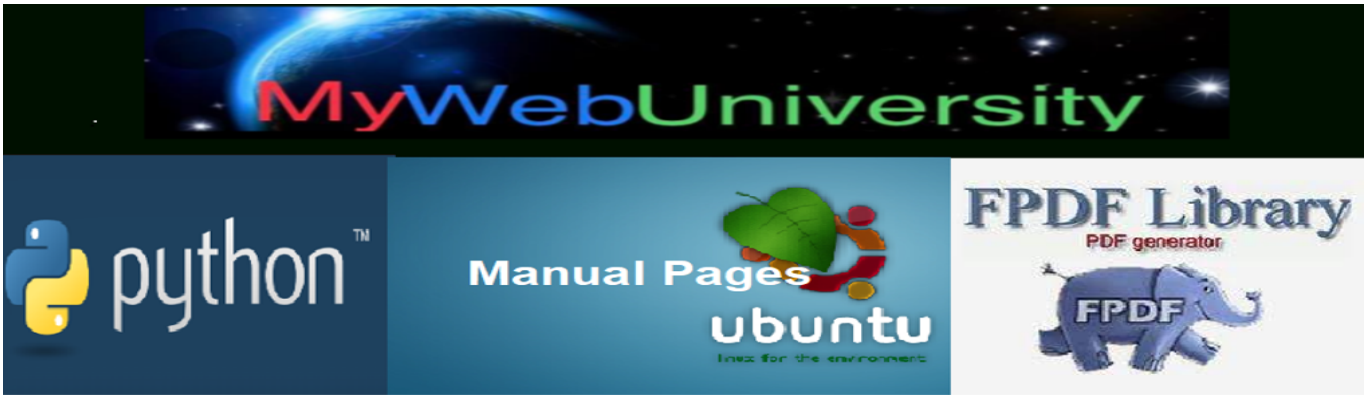

Full credit is given to the above companies including the Operating System (OS) that this PDF file was generated!

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

$ man access.2

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

NAME

access, faccessat, faccessat2 - check user's permissions for a file

SYNOPSIS

```
#include <unistd.h>

int access(const char *pathname, int mode);

#include <fcntl.h>           /* Definition of AT_* constants */

#include <unistd.h>

int faccessat(int dirfd, const char *pathname, int mode, int flags);

                /* But see C library/kernel differences, below */

int faccessat2(int dirfd, const char *pathname, int mode, int flags);
```

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

faccessat():

Since glibc 2.10:

_POSIX_C_SOURCE >= 200809L

Before glibc 2.10:

_ATFILE_SOURCE

DESCRIPTION

access() checks whether the calling process can access the file pathname. If pathname is a symbolic link, it is dereferenced.

The mode specifies the accessibility check(s) to be performed, and is either the value F_OK, or a mask consisting of the bitwise OR of one or more of R_OK, W_OK, and X_OK. F_OK tests for the existence of the file. R_OK, W_OK, and X_OK test whether the file exists

and grants read, write, and execute permissions, respectively.

The check is done using the calling process's real UID and GID, rather than the effective IDs as is done when actually attempting an operation (e.g., open(2)) on the file. Similarly, for the root user, the check uses the set of permitted capabilities rather than the set of effective capabilities; and for non-root users, the check uses an empty set of capabilities.

This allows set-user-ID programs and capability-endowed programs to easily determine the invoking user's authority. In other words, access() does not answer the "can I read/write/execute this file?" question. It answers a slightly different question: "(assuming I'm a setuid binary) can the user who invoked me read/write/execute this file?", which gives set-user-ID programs the possibility to prevent malicious users from causing them to read files which users shouldn't be able to read.

If the calling process is privileged (i.e., its real UID is zero), then an X_OK check is successful for a regular file if execute permission is enabled for any of the file owner, group, or other.

faccessat()

faccessat() operates in exactly the same way as access(), except for the differences described here.

If the pathname given in pathname is relative, then it is interpreted relative to the directory referred to by the file descriptor dirfd (rather than relative to the current working directory of the calling process, as is done by access() for a relative pathname).

If pathname is relative and dirfd is the special value AT_FDCWD, then pathname is interpreted relative to the current working directory of the calling process (like access()).

If pathname is absolute, then dirfd is ignored.

flags is constructed by ORing together zero or more of the following values:

AT_EACCESS

Perform access checks using the effective user and group IDs. By default, faccessat() uses the real IDs (like access()).

AT_SYMLINK_NOFOLLOW

If pathname is a symbolic link, do not dereference it: instead return information about the link itself.

See openat(2) for an explanation of the need for faccessat().

faccessat2()

The description of faccessat() given above corresponds to POSIX.1 and to the implementa? tion provided by glibc. However, the glibc implementation was an imperfect emulation (see BUGS) that papered over the fact that the raw Linux faccessat() system call does not have a flags argument. To allow for a proper implementation, Linux 5.8 added the faccessat2() system call, which supports the flags argument and allows a correct implementation of the faccessat() wrapper function.

RETURN VALUE

On success (all requested permissions granted, or mode is F_OK and the file exists), zero is returned. On error (at least one bit in mode asked for a permission that is denied, or mode is F_OK and the file does not exist, or some other error occurred), -1 is returned, and errno is set appropriately.

ERRORS

access() and faccessat() shall fail if:

EACCES The requested access would be denied to the file, or search permission is denied for one of the directories in the path prefix of pathname. (See also path_resolu? tion(7).)

ELOOP Too many symbolic links were encountered in resolving pathname.

ENAMETOOLONG

pathname is too long.

ENOENT A component of pathname does not exist or is a dangling symbolic link.

ENOTDIR

A component used as a directory in pathname is not, in fact, a directory.

EROFS Write permission was requested for a file on a read-only filesystem.

access() and faccessat() may fail if:

EFAULT pathname points outside your accessible address space.

EINVAL mode was incorrectly specified.

EIO An I/O error occurred.

ENOMEM Insufficient kernel memory was available.

ETXTBSY

Write access was requested to an executable which is being executed.

The following additional errors can occur for faccessat():

EBADF dirfd is not a valid file descriptor.

EINVAL Invalid flag specified in flags.

ENOTDIR

pathname is relative and dirfd is a file descriptor referring to a file other than a directory.

VERSIONS

faccessat() was added to Linux in kernel 2.6.16; library support was added to glibc in version 2.4.

faccessat2() was added to Linux in version 5.8.

CONFORMING TO

access(): SVr4, 4.3BSD, POSIX.1-2001, POSIX.1-2008.

faccessat(): POSIX.1-2008.

faccessat2(): Linux-specific.

NOTES

Warning: Using these calls to check if a user is authorized to, for example, open a file before actually doing so using open(2) creates a security hole, because the user might ex? ploit the short time interval between checking and opening the file to manipulate it. For this reason, the use of this system call should be avoided. (In the example just de? scribed, a safer alternative would be to temporarily switch the process's effective user ID to the real ID and then call open(2).)

access() always dereferences symbolic links. If you need to check the permissions on a symbolic link, use faccessat() with the flag AT_SYMLINK_NOFOLLOW.

These calls return an error if any of the access types in mode is denied, even if some of the other access types in mode are permitted.

If the calling process has appropriate privileges (i.e., is superuser), POSIX.1-2001 per? mits an implementation to indicate success for an X_OK check even if none of the execute file permission bits are set. Linux does not do this.

A file is accessible only if the permissions on each of the directories in the path prefix of pathname grant search (i.e., execute) access. If any directory is inaccessible, then the access() call fails, regardless of the permissions on the file itself.

Only access bits are checked, not the file type or contents. Therefore, if a directory is found to be writable, it probably means that files can be created in the directory, and not that the directory can be written as a file. Similarly, a DOS file may be found to be "executable," but the execve(2) call will still fail.

These calls may not work correctly on NFSv2 filesystems with UID mapping enabled, because

UID mapping is done on the server and hidden from the client, which checks permissions. (NFS versions 3 and higher perform the check on the server.) Similar problems can occur to FUSE mounts.

C library/kernel differences

The raw faccessat() system call takes only the first three arguments. The AT_EACCESS and AT_SYMLINK_NOFOLLOW flags are actually implemented within the glibc wrapper function for faccessat(). If either of these flags is specified, then the wrapper function employs fs?tatat(2) to determine access permissions, but see BUGS.

Glibc notes

On older kernels where faccessat() is unavailable (and when the AT_EACCESS and AT_SYM?LINK_NOFOLLOW flags are not specified), the glibc wrapper function falls back to the use of access(). When pathname is a relative pathname, glibc constructs a pathname based on the symbolic link in /proc/self/fd that corresponds to the dirfd argument.

BUGS

Because the Linux kernel's faccessat() system call does not support a flags argument, the glibc faccessat() wrapper function provided in glibc 2.32 and earlier emulates the re?quired functionality using a combination of the faccessat() system call and fstatat(2). However, this emulation does not take ACLs into account. Starting with glibc 2.33, the wrapper function avoids this bug by making use of the faccessat2() system call where it is provided by the underlying kernel.

In kernel 2.4 (and earlier) there is some strangeness in the handling of X_OK tests for superuser. If all categories of execute permission are disabled for a nondirectory file, then the only access() test that returns -1 is when mode is specified as just X_OK; if R_OK or W_OK is also specified in mode, then access() returns 0 for such files. Early 2.6 kernels (up to and including 2.6.3) also behaved in the same way as kernel 2.4.

In kernels before 2.6.20, these calls ignored the effect of the MS_NOEXEC flag if it was used to mount(2) the underlying filesystem. Since kernel 2.6.20, the MS_NOEXEC flag is honored.

SEE ALSO

chmod(2), chown(2), open(2), setgid(2), setuid(2), stat(2), euidaccess(3), credentials(7), path_resolution(7), symlink(7)

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/.