16

EXTENDED ATTRIBUTES

This chapter describes extended attributes (EAs), which allow arbitrary metadata, in the form of name-value pairs, to be associated with file i-nodes. EAs were added to Linux in version 2.6.

16.1 Overview

EAs are used to implement access control lists (Chapter 17) and file capabilities (Chapter 39). However, the design of EAs is general enough to allow them to be used for other purposes as well. For example, EAs could be used to record a file version number, information about the MIME type or character set for the file, or (a pointer to) a graphical icon.

EAs are not specified in SUSv3. However, a similar feature is provided on a few other UNIX implementations, notably the modern BSDs (see *extattr(2)*) and Solaris 9 and later (see *fsattr(5)*).

EAs require support from the underlying file system. This support is provided in *Btrfs*, *ext2*, *ext3*, *ext4*, *JFS*, *Reiserfs*, and *XFS*.

> Support for EAs is optional for each file system, and is controlled by kernel configuration options under the *File systems* menu. EAs are supported on *Reiserfs* since Linux 2.6.7.

EA namespaces

EAs have names of the form *namespace.name*. The *namespace* component serves to separate EAs into functionally distinct classes. The *name* component uniquely identifies an EA within the given *namespace*.

Four values are supported for *namespace*: *user*, *trusted*, *system*, and *security*. These four types of EAs are used as follows:

- *User* EAs may be manipulated by unprivileged processes, subject to file permission checks: to retrieve the value of a *user* EA requires read permission on the file; to change the value of a *user* EA requires write permission. (Lack of the required permission results in an `EACCES` error.) In order to associate *user* EAs with a file on *ext2*, *ext3*, *ext4*, or *Reiserfs* file systems, the underlying file system must be mounted with the *user_xattr* option:

```
$ mount -o user_xattr device directory
```

- *Trusted* EAs are like *user* EAs in that they can be manipulated by user processes. The difference is that a process must be privileged (`CAP_SYS_ADMIN`) in order to manipulate *trusted* EAs.
- *System* EAs are used by the kernel to associate system objects with a file. Currently, the only supported object type is an access control list (Chapter 17).
- *Security* EAs are used to store file security labels for operating system security modules, and to associate capabilities with executable files (Section 39.3.2). *Security* EAs were initially devised to support Security-Enhanced Linux (SELinux, *http://www.nsa.gov/research/selinux/*).

An i-node may have multiple associated EAs, in the same namespace or in different namespaces. The EA names within each namespace are distinct sets. In the *user* and *trusted* namespaces, EA names can be arbitrary strings. In the *system* namespace, only names explicitly permitted by the kernel (e.g., those used for access control lists) are allowed.

> *JFS* supports another namespace, *os2*, that is not implemented in other file systems. The *os2* namespace is provided to support legacy OS/2 file-system EAs. A process doesn't need to be privileged in order to create *os2* EAs.

Creating and viewing EAs from the shell

From the shell, we can use the *setfattr(1)* and *getfattr(1)* commands to set and view the EAs on a file:

```
$ touch tfile
$ setfattr -n user.x -v "The past is not dead." tfile
$ setfattr -n user.y -v "In fact, it's not even past." tfile
$ getfattr -n user.x tfile          Retrieve value of a single EA
# file: tfile                       Informational message from getfattr
user.x="The past is not dead."      The getfattr command prints a blank
                                    line after each file's attributes
$ getfattr -d tfile                 Dump values of all user EAs
# file: tfile
```

```
user.x="The past is not dead."
user.y="In fact, it's not even past."

$ setfattr -n user.x tfile          Change value of EA to be an empty string
$ getfattr -d tfile
# file: tfile
user.x
user.y="In fact, it's not even past."

$ setfattr -x user.y tfile          Remove an EA
$ getfattr -d tfile
# file: tfile
user.x
```

One of the points that the preceding shell session demonstrates is that the value of an EA may be an empty string, which is not the same as an EA that is undefined. (At the end of the shell session, the value of *user.x* is an empty string and *user.y* is undefined.)

By default, *getfattr* lists only the values of *user* EAs. The *−m* option can be used to specify a regular expression pattern that selects the EA names that are to be displayed:

```
$ getfattr -m 'pattern' file
```

The default value for *pattern* is `^user\.`. We can list all EAs on a file using the following command:

```
$ getfattr -m - file
```

16.2 Extended Attribute Implementation Details

In this section, we extend the overview of the preceding section to fill in a few details of the implementation of EAs.

Restrictions on *user* extended attributes

It is only possible to place *user* EAs on files and directories. Other file types are excluded for the following reasons:

- For a symbolic link, all permissions are enabled for all users, and these permissions can't be changed. (Symbolic link permissions have no meaning on Linux, as detailed in Section 18.2.) This means that permissions can't be used to prevent arbitrary users from placing *user* EAs on a symbolic link. The resolution of this problem is to prevent all users from creating *user* EAs on the symbolic link.
- For device files, sockets, and FIFOs, the permissions control the access that users are granted for the purpose of performing I/O on the underlying object. Manipulating these permissions to control the creation of *user* EAs would conflict with this purpose.

Furthermore, it is not possible for an unprivileged process to place a *user* EA on a directory owned by another user if the sticky bit (Section 15.4.5) is set on the directory.

This prevents arbitrary users from attaching EAs to directories such as `/tmp`, which are publicly writable (and so would allow arbitrary users to manipulate EAs on the directory), but which have the sticky bit set to prevent users from deleting files owned by other users in the directory.

Implementation limits

The Linux VFS imposes the following limits on EAs on all file systems:

- The length of an EA name is limited to 255 characters.
- An EA value is limited to 64 kB.

In addition, some file systems impose more restrictive limits on the size and number of EAs that can be associated with a file:

- On *ext2*, *ext3*, and *ext4*, the total bytes used by the names and values of all EAs on a file is limited to the size of a single logical disk block (Section 14.3): 1024, 2048, or 4096 bytes.
- On *JFS*, there is an upper limit of 128 kB on the total bytes used by the names and values of all EAs on a file.

16.3 System Calls for Manipulating Extended Attributes

In this section, we look at the system calls used to update, retrieve, and remove EAs.

Creating and modifying EAs

The *setxattr()*, *lsetxattr()*, and *fsetxattr()* system calls set the value of one of a file's EAs.

```
#include <sys/xattr.h>

int setxattr(const char *pathname, const char *name, const void *value,
             size_t size, int flags);
int lsetxattr(const char *pathname, const char *name, const void *value,
             size_t size, int flags);
int fsetxattr(int fd, const char *name, const void *value,
             size_t size, int flags);
```

All return 0 on success, or −1 on error

The differences between these three calls are analogous to those between *stat()*, *lstat()*, and *fstat()* (Section 15.1):

- *setxattr()* identifies a file by *pathname*, and dereferences the filename if it is a symbolic link;
- *lsetxattr()* identifies a file by *pathname*, but doesn't dereference symbolic links; and
- *fsetxattr()* identifies a file by the open file descriptor *fd*.

The same distinction applies to the other groups of system calls described in the remainder of this section.

The *name* argument is a null-terminated string that defines the name of the EA. The *value* argument is a pointer to a buffer that defines the new value for the EA. The *size* argument specifies the length of this buffer.

By default, these system calls create a new EA if one with the given *name* doesn't already exist, or replace the value of an EA if it does already exist. The *flags* argument provides finer control over this behavior. It may be specified as 0 to obtain the default behavior, or as one of the following constants:

`XATTR_CREATE`
: Fail (`EEXIST`) if an EA with the given *name* already exists.

`XATTR_REPLACE`
: Fail (`ENODATA`) if an EA with the given *name* doesn't already exist.

Here an example of the use of *setxattr()* to create a *user* EA:

```
char *value;

value = "The past is not dead.";

if (setxattr(pathname, "user.x", value, strlen(value), 0) == -1)
    errExit("setxattr");
```

Retrieving the value of an EA

The *getxattr()*, *lgetxattr()*, and *fgetxattr()* system calls retrieve the value of an EA.

```
#include <sys/xattr.h>

ssize_t getxattr(const char *pathname, const char *name, void *value,
                 size_t size);
ssize_t lgetxattr(const char *pathname, const char *name, void *value,
                 size_t size);
ssize_t fgetxattr(int fd, const char *name, void *value,
                 size_t size);
```

All return (nonnegative) size of EA value on success, or –1 on error

The *name* argument is a null-terminated string that identifies the EA whose value we want to retrieve. The EA value is returned in the buffer pointed to by *value*. This buffer must be allocated by the caller, and its length must be specified in *size*. On success, these system calls return the number of bytes copied into *value*.

If the file doesn't have an attribute with the given *name*, these system calls fail with the error `ENODATA`. If *size* is too small, these system calls fail with the error `ERANGE`.

It is possible to specify *size* as 0, in which case *value* is ignored but the system call still returns the size of the EA value. This provides a mechanism to determine the size of the *value* buffer required for a subsequent call to actually retrieve the EA value. Note, however, that we still have no guarantee that the returned size will be big enough when subsequently trying to retrieve the value. Another process may have assigned a bigger value to the attribute in the meantime, or removed the attribute altogether.

Removing an EA

The *removexattr()*, *lremovexattr()*, and *fremovexattr()* system calls remove an EA from a file.

```
#include <sys/xattr.h>

int removexattr(const char *pathname, const char *name);
int lremovexattr(const char *pathname, const char *name);
int fremovexattr(int fd, const char *name);
```

All return 0 on success, or −1 on error

The null-terminated string given in *name* identifies the EA that is to be removed. An attempt to remove an EA that doesn't exist fails with the error ENODATA.

Retrieving the names of all EAs associated with a file

The *listxattr()*, *llistxattr()*, and *flistxattr()* system calls return a list containing the names of all of the EAs associated with a file.

```
#include <sys/xattr.h>

ssize_t listxattr(const char *pathname, char *list, size_t size);
ssize_t llistxattr(const char *pathname, char *list, size_t size);
ssize_t flistxattr(int fd, char *list, size_t size);
```

All return number of bytes copied into *list* on success, or −1 on error

The list of EA names is returned as a series of null-terminated strings in the buffer pointed to by *list*. The size of this buffer must be specified in *size*. On success, these system calls return the number of bytes copied into *list*.

As with *getxattr()*, it is possible to specify *size* as 0, in which case *list* is ignored, but the system call returns the size of the buffer that would be required for a subsequent call to actually retrieve the EA name list (assuming it remains unchanged).

To retrieve a list of the EA names associated with a file requires only that the file be accessible (i.e., that we have execute access to all of the directories included in *pathname*). No permissions are required on the file itself.

For security reasons, the EA names returned in *list* may exclude attributes to which the calling process doesn't have access. For example, most file systems omit *trusted* attributes from the list returned by a call to *listxattr()* in an unprivileged process. But note the "may" in the earlier sentence, indicating that a file-system implementation is not obliged to do this. Therefore, we need to allow for the possibility that a subsequent call to *getxattr()* using an EA name returned in *list* may fail because the process doesn't have the privilege required to obtain the value of that EA. (A similar failure could also happen if another process deleted an attribute between the calls to *listxattr()* and *getxattr()*.)

Example program

The program in Listing 16-1 retrieves and displays the names and values of all EAs of the files listed on its command line. For each file, the program uses *listxattr()* to retrieve the names of all EAs associated with the file, and then executes a loop calling *getxattr()* once for each name, to retrieve the corresponding value. By default, attribute values are displayed as plain text. If the *–x* option is supplied, then the attribute values are displayed as hexadecimal strings. The following shell session log demonstrates the use of this program:

```
$ setfattr -n user.x -v "The past is not dead." tfile
$ setfattr -n user.y -v "In fact, it's not even past." tfile
$ ./xattr_view tfile
tfile:
        name=user.x; value=The past is not dead.
        name=user.y; value=In fact, it's not even past.
```

Listing 16-1: Display file extended attributes

xattr/xattr_view.c

```
#include <sys/xattr.h>
#include "tlpi_hdr.h"

#define XATTR_SIZE 10000

static void
usageError(char *progName)
{
    fprintf(stderr, "Usage: %s [-x] file...\n", progName);
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    char list[XATTR_SIZE], value[XATTR_SIZE];
    ssize_t listLen, valueLen;
    int ns, j, k, opt;
    Boolean hexDisplay;

    hexDisplay = 0;
    while ((opt = getopt(argc, argv, "x")) != -1) {
        switch (opt) {
        case 'x': hexDisplay = 1;       break;
        case '?': usageError(argv[0]);
        }
    }

    if (optind >= argc + 2)
        usageError(argv[0]);
```

```
    for (j = optind; j < argc; j++) {
        listLen = listxattr(argv[j], list, XATTR_SIZE);
        if (listLen == -1)
            errExit("listxattr");

        printf("%s:\n", argv[j]);

        /* Loop through all EA names, displaying name + value */

        for (ns = 0; ns < listLen; ns += strlen(&list[ns]) + 1) {
            printf("        name=%s; ", &list[ns]);

            valueLen = getxattr(argv[j], &list[ns], value, XATTR_SIZE);
            if (valueLen == -1) {
                printf("couldn't get value");
            } else if (!hexDisplay) {
                printf("value=%.*s", (int) valueLen, value);
            } else {
                printf("value=");
                for (k = 0; k < valueLen; k++)
                    printf("%02x ", (unsigned int) value[k]);
            }

            printf("\n");
        }

        printf("\n");
    }

    exit(EXIT_SUCCESS);
}
```

xattr/xattr_view.c

16.4 Summary

From version 2.6 onward, Linux supports extended attributes, which allow arbitrary metadata to be associated with a file, in the form of name-value pairs.

16.5 Exercise

16-1. Write a program that can be used to create or modify a *user* EA for a file (i.e., a simple version of *setfattr(1)*). The filename and the EA name and value should be supplied as command-line arguments to the program.