A Bug in Linux ACL Implementation

Jaroslav Janáček

Comenius University in Bratislava, Faculty of Mathematics, Physics and Informatics, Department of Computer Science, Mlynská dolina, 842 48, Bratislava, Slovakia

Abstract

We demonstrate and analyse a long-standing bug in the Linux kernel ACL permission checking code that, under specific circumstances, allows users and/or groups to access filesystem objects they should not be allowed to access and propose a fix.

Keywords

Linux, ACL

1. Introduction

Access control, controlling access to filesystem objects, is an important part of operating systems. In Linux based operating systems, the filesystem access control is done in the Linux kernel. The Linux kernel's basic access control model is based on UNIX access control model, where the permissions to read, write, and execute a file can be assigned to the owner of the file, a single group, and others. While this basic model is sufficient in many simple cases, there are cases when we need to assign different permissions to different users and/or groups. For example, if a file is to be readable and writeable by one group of users, and readable only by another group of users, this cannot be achieved using this simple model in a simple way (in general). To enhance the capabilities of access control, the Linux kernel extends the basic model with *Access Control Lists (ACL)*. An ACL associated with a file can be used to assign permissions to additional users and/or groups.

When experimenting with ACLs in Linux we have observed a strange behaviour in some corner cases. Because it can compromise security, we consider it to be a bug. We present our analysis of the problem and propose a fix in this paper.

2. Linux ACLs

The Linux kernel allows an ACL to be associated with any filesystem object residing in a filesystem that supports ACL storage (e.g. **ext4** and many other commonly used filesystems for Linux). A directory can also have a *default ACL* associated with it. The default ACL is used as a template for initialization of the ACL of a new object when it is created in the directory.

Each ACL consists of ACL entries. There are six types of ACL entries:

- ACL_USER_OBJ – specifies the permissions for the object's owner,
- ACL_USER – specifies the permissions for a specific user,
- ACL_GROUP_OBJ – specifies the permissions for the object's group,
- ACL_GROUP – specifies the permissions for a specific group,
- ACL_MASK – specifies the upper limit of permissions granted by the entries of the type ACL_USER, ACL_GROUP, or ACL_GROUP_OBJ,
- ACL_OTHER – specifies the permissions for others.

Each ACL must contain exactly one entry of each of the types ACL_USER_OBJ, ACL_GROUP_OBJ, and ACL_OTHER. It may contain zero or more entries of each of the types ACL_USER and ACL_GROUP, and it may contain zero or one entry of the type ACL_MASK. If it contains an entry of the type ACL_USER or ACL_GROUP, it must contain an entry of the type ACL_MASK. Each ACL entry of the type ACL_USER or ACL_GROUP contains a user/group ID.

A process runs on behalf of a user who is a member of a primary group, and who can be a member of a number of supplementary groups. The user is identified by the *effective user ID* and the primary group is identified by the *effective group ID*. To be correct, we must say that in the Linux kernel the *filesystem user ID* and the *filesystem group ID* are actually used instead, but these are usually equal to the effective user ID and the effective group ID, so we will not distinguish among them.

Each filesystem object is assigned an owner, identified by an user ID, and a group, identified by a group ID. It is also assigned a 12-bit vector of UNIX permissions – 3 bits (read, write, execute) for the owner, the group, and others, and 3 special bits (set-user-ID, set-group-ID, and sticky bit).

ITAT'24: Information technologies – Applications and Theory, September 20–24, 2024, Drienica, Slovakia

jaroslav.janacek@uniba.sk (J. Janáček)

When a process tries to open a filesystem object which has an ACL associated with it, the Linux kernel performs a permission check algorithm, which should work as follows[1]:

1. If the effective user ID of the process matches the user ID of the object's owner, the ACL_USER_OBJ entry is used – if it contains the requested permissions, the access is granted, otherwise the access is denied.
2. Otherwise, if the effective user ID matches the user ID of an entry of the type ACL_USER, this entry is used – if the entry and the ACL_MASK entry both contain the requested permissions, the access is granted, otherwise the access is denied.
3. Otherwise, if the effective group ID or any supplementary group ID match the group ID of the object or the group ID of an ACL entry of the type ACL_GROUP, then these matching ACL entries (of the types ACL_GROUP_OBJ and/or ACL_GROUP) are used – if the ACL entry of the type ACL_MASK and any of the matching ACL entries both contain the requested permissions, the access if granted, otherwise it is denied. If there is no entry of the type ACL_MASK, the only matching ACL entry can be the one of the type ACL_GROUP_OBJ, and that is used alone.
4. Otherwise, if the entry of the type ACL_OTHER contains the requested permissions, the access is granted.
5. Otherwise, the access is denied.

There is an exception, if the process has a special effective capability to override the access control (typically when it runs on behalf of the root user – effective user ID zero), the permission check algorithm is skipped.

As we can see, in all cases the effective permissions are determined either by a single ACL entry (ACL_USER_OBJ, ACL_OTHER, or ACL_GROUP_OBJ if there is no ACL_MASK entry), or by the logical AND of a single ACL entry (ACL_USER, ACL_GROUP, ACL_GROUP_OBJ) and the ACL_MASK entry. Also, if the effective user ID of the process matches either the owner's user ID, or the user ID in any of the ACL_USER entries, the group ACL entries and the ACL_OTHER entry are not used. And if the effective group ID or any supplementary group ID match the object's group ID or the group ID of any ACL_GROUP entry, the ACL_OTHER entry is not used. In other words, the ACL_OTHER entry can only be used for processes running on behalf of a user who has no matching ACL entries – neither directly, nor via a group membership.

2.1. Correspondence between ACL entries and standard UNIX permissions

As we have mentioned above, the Linux kernel does not replace the standard UNIX permissions assigned to filesystem objects with ACLs, but ACLs are an extension. If there is no ACL associated with a file, the standard UNIX permissions are used; if there is an ACL associated with the file, the ACL is used. There are, however, many programs that do not know about ACLs (e.g. because they had been written before ACLs were widely supported). When dealing with security, it is very important not to break things people rely on. This is why the interaction of standard UNIX system calls and utilities dealing with permissions and ACLs is important.

A minimal ACL consists of the three mandatory entries – ACL_USER_OBJ, ACL_GROUP_OBJ, and ACL_OTHER. The meaning of these entries, in this simple case, matches exactly the meaning of the standard permission bits for the owner, for the object's group, and for others. Indeed, if we change an ACL entry, the corresponding permission bits are changed as well, and vice versa. This ensures compatibility with legacy tools in this simple case.

When an entry of another type is added to the ACL, things get a little bit more complicated. The permission bits for the owner and for others still correspond to the ACL_USER_OBJ and ACL_OTHER entries, but the standard permission bits for the object's group now correspond to the permissions of the ACL_MASK entry. While this may sound a bit strange at first, there is a good reason behind it. Imagine a legacy program that wants to make sure that only the owner of a file can access it. Such program would change the standard permission bits for the object's group and for others to zero, leaving only some nonzero bits for the owner. If the object's group permission bits corresponded to the ACL_GROUP_OBJ entry while there was an ACL_USER or ACL_GROUP type ACL entry present in the ACL, the legacy program would not remove the permissions granted by these ACL entries. But, because the standard object's group permission bits correspond to the ACL_MASK entry, if it is present, zeroing the standard object's group permission bits zeroes the ACL_MASK entry, thereby effectively removing all permissions granted by ACL_USER, ACL_GROUP, and ACL_GROUP_OBJ entries. This way, a sort of compatibility with legacy tools is ensured even in the more complex cases.

3. Problem demonstration

We will now show that there are cases when the system does not behave according to the permission check algorithm as described above. We will use the kernel version 6.1, however, the same behaviour can be found in many

previous versions as well (5.x, 4.x, 3.x at least).

3.1. Problem with ACL_USER entries

First, we create a file *f* and set it's ACL so that only the file's owner (*jerry*) has the read and write permissions, and others have the read permission:

```
jerry: echo 'Works!' > f
jerry: setfacl -m 'u::rw,g::-,o::r' f
jerry: getfacl f
# file: f
# owner: jerry
# group: jerry
user::rw-
group::---
other::r--
```

The `setfacl` command is used to modify (with the `-m` flag) the ACL, the `getfacl` command is used to show the current ACL associated with the file.

Clearly, at this stage, another user (*tom*), who is not a member of the group *jerry*, should be able to read the file, because there are no ACL entries matching *tom* or any of his groups, and the ACL_OTHER entry grants the read permission to others. We test it by trying to show the contents of the file as *tom*, and also by trying to modify the file:

```
tom: cat f
Works!
tom: echo aaa >> f
bash: f: Permission denied
```

Now, the owner of the file (*jerry*) modifies the ACL so that the user *tom* is granted no permissions:

```
jerry: setfacl -m 'u:tom:-' f
jerry: getfacl f
# file: f
# owner: jerry
# group: jerry
user::rw-
user:tom:---
group::---
mask::---
other::r--
```

As we can see, two new ACL entries have been created - the ACL_USER entry for the user *tom*, and the ACL_MASK entry (which must be in the ACL when there is any ACL_USER or ACL_GROUP entry). The permissions of the (explicitly unspecified) ACL_MASK entry have been automatically calculated by the `setfacl` utility as the union (logical OR) of the permissions of all affected ACL entries (ACL_USER, ACL_GROUP, ACL_GROUP_OBJ types).

According to the permission check algorithm described in the previous section, the user *tom* should have no access to the file. However, when we try it:

```
tom: cat f
Works!
tom: echo aaa >> f
bash: f: Permission denied
```

we can see, the user can read the file, and cannot write to the file. This behaviour is obviously incorrect and may be considered a security problem.

It seems that the permissions for others are used for the user *tom* in this case, although, according to the permission check algorithm, they should not be used. We can try to support this hypothesis by changing the permissions for others and retesting the *tom*'s access:

```
jerry: setfacl -m 'o::w' f
jerry: getfacl f
# file: f
# owner: jerry
# group: jerry
user::rw-
user:tom:---
group::---
mask::---
other::-w-

tom: cat f
cat: f: Permission denied
tom: echo aaa >> f

jerry: cat f
Works!
aaa
```

As we can see, after changing the permissions for others to write only, the user *tom* cannot read the file, but can successfully write to the file.

We can now change the permissions of the ACL_MASK entry in the ACL to a nonzero value, e.g. to read and write, and retest the *tom*'s access:

```
jerry: setfacl -m 'm::rw' f
jerry: getfacl f
# file: f
# owner: jerry
# group: jerry
user::rw-
user:tom:---
group::---
mask::rw-
other::-w-

tom: cat f
cat: f: Permission denied
```

```
tom: echo aaa >> f
bash: f: Permission denied
```

The system now behaves as expected – the user *tom* is denied both read and write access to the file.

3.2. Problem with ACL_GROUP entries

We can now repeat the tests with group permissions instead of user permissions. First we restore the ACL to the initial state:

```
jerry: setfacl -b f
jerry: setfacl -m 'o::r' f
jerry: getfacl f
# file: f
# owner: jerry
# group: jerry
user::rw-
group::---
other::r--
```

The `setfacl -b` command removes all ACL entries, and then we restore the read permissions for others.

The user *tom* should have the read access now again because there are no matching ACL entries for *tom* or his groups:

```
tom: cat f
Works!
aaa
tom: echo bbb >> f
bash: f: Permission denied
```

We add an entry for the group *users* with no permissions:

```
jerry: setfacl -m 'g:users:-' f
jerry: getfacl f
# file: f
# owner: jerry
# group: jerry
user::rw-
group::---
group:users:---
mask::---
other::r--
```

Again, two new entries have been created – the ACL_GROUP entry for the group *users* and the ACL_MASK entry. We can test the *tom*'s access:

```
tom: cat f
Works!
aaa
tom: echo bbb >> f
bash: f: Permission denied
```

We can see that the system behaves incorrectly again – the user *tom* as a member of the group *users* should have no access to the file, but the permissions for others seem to be applied instead.

When we change the permissions of the ACL_MASK entry to a nonzero value, it works correctly again:

```
jerry: setfacl -m 'm::rw' f
jerry: getfacl f
# file: f
# owner: jerry
# group: jerry
user::rw-
group::---
group:users:---
mask::rw-
other::r--

tom: cat f
cat: f: Permission denied
tom: echo bbb >> f
bash: f: Permission denied
```

4. Problem analysis

We have demonstrated the incorrect behaviour in the previous section and we have stated a hypothesis that under some conditions the ACL_OTHER entry (or perhaps the standard permission bits for others) are used instead of the correct ACL_USER or ACL_GROUP entry. As far as the conditions are concerned, it appears that this incorrect behaviour is manifested when the ACL_MASK entry contains empty permissions. In this section we will identify the root cause of this behaviour and refine and confirm our hypothesis.

To start, we look for the code responsible for the ACL permission check algorithm. After looking at the structure of the Linux kernel source code [2] we can easily discover an interesting file *fs/posix_acl.c*, and more specificaly the *posix_acl_permission* function. Quick analysis of this function leads to the conclusion that it does indeed implement the permission check algorithm in accordance with the specification in the section 2.

Looking for function calls of posix_acl_permissions leads to the *namei.c* file where we discover a chain of functions dealing with permission checking. A short description of the functions and their relationship is in the table 1.

The function *acl_permission_check*, shown in the listing 1, is of a particular interest. In the lines 7–13 the code checks if the effective user ID (or more precisely the filesystem user ID) matches the owner's user ID, and if it does, it uses the standard UNIX permission bits for the owner to determine whether to grant or deny the access.

Table 1
The functions dealing with permission checking

Function	Called from	Short description
posix_acl_permission	check_acl	Parse the ACL and perform the ACL permission check algorithm.
check_acl	acl_permission_check	Retrieve the ACL and call posix_acl_permission.
acl_permission_check	generic_permission	Handle the standard UNIX permission bits and call check_acl in case an ACL is present.
generic_permission	do_inode_permission	Call acl_permission_check, if it denies access, handle overrides (exceptions) based on capabilities.
do_inode_permission	inode_permission	Call generic_permission or a special function to check permissions provided by the filesystem.
inode_permission	may_open (and others)	Handle denying write access to immutable files, call do_inode_permission, call advanced security modules' hooks for additional restrictions.
may_open	do_open, vfs_tmpfile	Handle additional restrictions, call inode_permission.
do_open		Handles the last step of opening a file, including calling may_open to check if it is permitted.

In this case it entirely bypasses the ACL permission check algorithm, but it should not be a problem unless the ACL and the standard UNIX permission bits are desynchronized (which should not happen under normal operating conditions).

The lines 15–20 handle the case when an ACL is present – we will look into it in more detail in a short while.

The lines 22–34 handle the standard UNIX permissions for the object's group. We can see an optimization here – if the relevant permission bits for the object's group are equal to the relevant permission bits for others, the code skips checking if the object's group ID matches the process's group IDs and continues with checking the permission bits for others. It can do so, because the access would be granted or denied identically for both the object's group and others in this case.

Finally, the lines 36–37 check the permissions for others.

Let's now analyse the lines 15–20 in more detail. The line 16 is crucial here. It checks if the file has an ACL associated with it and if the standard permission bits contain at least one nonzero bit for the object's group permissions. If both conditions are true, the function *check_acl* is called to process the ACL, and the result is returned. If there is no ACL associated with the file, it makes no sense to call *check_acl*. But what is the purpose of the second condition? It clearly causes the observed incorrect behaviour – if the ACL_MASK entry contains no permissions, the standard UNIX permissions bits for the object's group are zero, the ACL processing is skipped, and the code continues with checking only the standard UNIX permission bits. And because in our test cases the process's group ID did not match the object's group ID, the standard UNIX permission bits for others were used to determine the access.

If the line 16 was changed to

```
if (IS_POSIXACL(inode)) {
```

everything would work just fine. The most probable reason for the second condition is an optimization. If the ACL_MASK (and therefore the standard object's group permission bits) contains no permissions, no permissions can be effectively granted using the ACL_USER or ACL_GROUP entries, and therefore the code's author has probably incorrectly concluded that is not necessary to process the ACL. But this is wrong – if the ACL contains an entry of the type ACL_USER or ACL_GROUP, it can still match and deny access.

5. Proposed fix

Having found the cause of the problem, we can propose two possible ways to fix it. One obvious fix is to change the line 16 of the function *acl_permission_check* in *fs/namei.c* as shown in the previous section. This would correct the problem but could slow down the evaluation of the function.

Another way, preserving the optimization in non-problematic cases, is to change the line 16 as follows:

Listing 1: fs/namei.c:acl_permission_check

```
1  static int acl_permission_check(struct user_namespace *mnt_userns,
2                                  struct inode *inode, int mask)
3  {
4          unsigned int mode = inode->i_mode;
5          kuid_t i_uid;
6
7          /* Are we the owner? If so, ACL's don't matter */
8          i_uid = i_uid_into_mnt(mnt_userns, inode);
9          if (likely(uid_eq(current_fsuid(), i_uid))) {
10                 mask &= 7;
11                 mode >>= 6;
12                 return (mask & ~mode) ? -EACCES : 0;
13         }
14
15         /* Do we have ACL's? */
16         if (IS_POSIXACL(inode) && (mode & S_IRWXG)) {
17                 int error = check_acl(mnt_userns, inode, mask);
18                 if (error != -EAGAIN)
19                         return error;
20         }
21
22         /* Only RWX matters for group/other mode bits */
23         mask &= 7;
24
25         /*
26          * Are the group permissions different from
27          * the other permissions in the bits we care
28          * about? Need to check group ownership if so.
29          */
30         if (mask & (mode ^ (mode >> 3))) {
31                 kgid_t kgid = i_gid_into_mnt(mnt_userns, inode);
32                 if (in_group_p(kgid))
33                         mode >>= 3;
34         }
35
36         /* Bits in 'mode' clear that we require? */
37         return (mask & ~mode) ? -EACCES : 0;
38 }
```

```
if (IS_POSIXACL(inode) && (mode & (
    S_IRWXG | S_IRWXO)) {
```

This would skip the (expensive) ACL processing in the case when there are no permissions in the standard UNIX permission bits for the object's group and in the standard UNIX permission bits for others. Assuming the standard UNIX permission bits are synchronized with the corresponding ACL entries, both the ACL_MASK and ACL_OTHER entries contain no permissions in this case, and therefore only the object's owner can have any permissions (which is not relevant at this point of the code).

The final question to ask is whether this optimization is actually worth it. The cases when both the ACL_MASK and ACL_OTHER entries of an ACL contain no permissions may be not so rare. They include the cases when a legacy tool removes all permissions except those for the object's owner.

6. Conclusions

We have identified a long-standing bug in the Linux kernel permission checking code. We have demonstrated the bug in several examples, identified the piece of kernel

code that causes it, and proposed two fixes with different performance impact.

At the time of writing this paper, we have rechecked the code in the newest available Linux kernel source code (6.10-rc7) and the relevant code is still unchanged.

We have also discovered that there are other cases when the ACL is not checked at all, but these should not cause any problems unless the standard UNIX permission bits and the corresponding ACL entries become desynchronized.

Unless we receive some negative feedback, we plan to submit one of the proposed fixes to the Linux kernel maintainers.

Acknowledgments

This publication is the result of support under the Operational Program Integrated Infrastructure for the project: Advancing University Capacity and Competence in Research, Development a Innovation (ACCORD, ITMS2014+:313021X329), co-financed by the European Regional Development Fund.

References

[1] A. Gruenbacher, acl – Access Control Lists, man 5 acl, 2002.

[2] L. Torvalds, et al., The Linux kernel source code, https://kernel.org/, 1991–2024. Accessed 2024-07-10.