Missing system calls in the Linux kernel audit subsystem classes (CVE-2025-71239, CVE-2026-23241)

Mar 08, 2026 in **HACKING** • **LINUX**
audit **auditd**
10 min read

Introduction

The Linux kernel audit subsystem is a security mechanism allowing one to detect potential threats on a Linux system. Its userland counterpart, `auditd` (the audit daemon), can be configured with different kind of rules. Part of these rules rely on hooking system calls in kernel land grouped by what are called *classes*. Some, relatively recent, system calls were missing from these classes, allowing one to bypass these type of `auditd` rules.

This article explains how these missing system calls were found and added to the kernel.

EDIT: This was attributed CVE-2025-71239 and CVE-2026-23241

auditd filesystem rules

The audit subsystem has a type of rules named *filesystem rules* or *watches*[1]:

> File System File System rules are sometimes called watches. These rules are used to audit access to particular files or directories that you may be interested in. If the path given in a watch rule is a directory, then the rule used is recursive to the bottom of the directory tree excluding any directories that may be mount points. The syntax of these watch rules generally follow this format:
>
> ```
> -w path-to-file -p permissions -k keyname
> ```
>
> where the permission are any one of the following:

```
      r - read of the file

      w - write to the file

      x - execute the file

      a - change in the file's attribute
```

The above *permissions* parameter is implemented using lists or *classes* of system calls to be hooked in kernel land in the kernel source code[2].

Each class contains a subset of the kernel system calls matching one of the *permissions*. For example, the `/include/asm-generic/audit_read.h` file defines what system calls triggers a “read permission”:

```
/* SPDX-License-Identifier: GPL-2.0 */
#ifdef __NR_readlink
__NR_readlink,
#endif
__NR_quotactl,
__NR_listxattr,
__NR_llistxattr,
__NR_flistxattr,
__NR_getxattr,
__NR_lgetxattr,
__NR_fgetxattr,
#ifdef __NR_readlinkat
__NR_readlinkat,
#endif
```

And the corresponding read class is defined in `/arch/x86/kernel/audit_64.c`:

```
static unsigned read_class[] = {
#include <asm-generic/audit_read.h>
~0U
};
...
static int __init audit_classes_init(void)
{
#ifdef CONFIG_IA32_EMULATION
    audit_register_class(AUDIT_CLASS_WRITE_32, ia32_write_class);
```

```
    audit_register_class(AUDIT_CLASS_READ_32, ia32_read_class);
...
```

Hence, the following `auditd` filesystem rule would be triggered if one uses `readlink()` or `listxattr()` from userland on the `/tmp/test.txt` file:

```
-w /tmp/test.txt -p r -k test_read
```

Raising an alert of type `SYSCALL`. Below is an example of such alert triggered by using the `ls` binary:

```
$ ls -lh /tmp/test.txt
-rw-rw-r-- 1 kali kali 0 Mar  8 06:53 /tmp/test.txt
# ausearch -k test_read | grep ls
type=SYSCALL msg=audit(1772967193.940:217): arch=c000003e syscall=194 success=yes exit=0 a
```

The `/usr/bin/ls` binary indeed make system calls contained in the `read_class` under the hood, for example, to check for SELinux extended attributes:

```
$ strace ls -lh /tmp/test.txt
...
lgetxattr("/tmp/test.txt", "security.selinux", 0x558df9de43c0, 255) = -1 ENODATA (No data
listxattr("/tmp/test.txt", "", 152)     = 0
...
write(1, "-rw-rw-r-- 1 kali kali 0 Mar  8 "..., 52-rw-rw-r-- 1 kali kali 0 Mar  8 06:53 /t
) = 52
...
+++ exited with 0 +++
```

Missing system calls in audit classes

System calls in the kernel gets added regularly, and the maintainers made a documentation on how this should be done[3]. The *other details* section of the documentation states that the audit subsystem should be updated in particular cases:

> ## Other Details

> Most of the kernel treats system calls in a generic way, but there is the occasional exception that may need updating for your particular system call.
>
> The audit subsystem is one such special case; it includes (arch-specific) functions that classify some special types of system call – specifically file open (`open`/`openat`), program execution (`execve`/`exeveat`) or socket multiplexor (`socketcall`) operations. If your new system call is analogous to one of these, then the audit system should be updated.
>
> More generally, if there is an existing system call that is analogous to your new system call, it's worth doing a kernel-wide grep for the existing system call to check there are no other special cases.

This implicitly means that when one adds a system call, it should also be added it in whatever audit subsystem class it may belong.

Looking into the kernel codebase to understand the audit subsystem implementation, I noticed the following system calls were missing from the aforementioned classes:

- `fchmodat2()`
- `getxattrat()`
- `listxattrat()`

The consequence of such oversights is that userland programs calling these particular system calls will not trigger `auditd` rules, allowing for detection bypasses.

fchmodat2

The `chmod()` system call is used to change the permissions of a file. Because of the lack of flexibility of this call, the subsequent `fchmodat()` call was added in kernel v2.6.16[4] and later on the `fchmodat2()` call in kernel v6.6[5].

While the `fchmodat()` system call was added to the `audit_change_attr.h` class file in the initial audit class definition[6], the `fchmodat2()` was later not added to it.

This can be verified with the following two programs, the first one using `fchmodat()` to change the permissions of a file given in argument:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
```

```c
#include <fcntl.h>           /* Definition of AT_* constants */
#include <sys/stat.h>

int main(int argc, char *argv[])
{
        int mode = strtol(argv[2], 0, 8);
        int status = fchmodat(0, argv[1], mode, 0);

        if (status == -1)
        {
                printf("fchmodat failed: %s\n", strerror(errno));
                return 1;
        }

        return 0;
}
```

and the second one using `fchmodat2()`:

```c
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/syscall.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
        int mode = strtol(argv[2], 0, 8);
        int status = syscall(SYS_fchmodat2, 0, argv[1], mode, 0);

        if (status == -1) {
                printf("fchmodat2 failed: %s\n", strerror(errno));
                return 1;
        }

        return 0;
}
```

If an `auditd` rule in `/etc/audit/rules.d/missing.rules` is defined as such:

```
-w /tmp/test.txt -p a -k test_change_attr
```

And the `fchmodat` program is executed against `/tmp/test.txt`:

```
$ ls -lh /tmp/test.txt
-rw-rw-r-- 1 kali kali 0 Mar  8 06:33 /tmp/test.txt
$ ./fchmodat /tmp/test.txt 777
$ ls -lh /tmp/test.txt
-rwxrwxrwx 1 kali kali 0 Mar  8 06:33 /tmp/test.txt
```

Then the rule gets triggered with a `SYSCALL` type alert, as expected:

```
# ausearch -k test_change_attr | grep fchmodat
type=SYSCALL msg=audit(1772966190.736:169): arch=c000003e syscall=268 success=yes exit=0 a
```

If now the `fchmodat2` program is executed against the same file:

```
$ ls -lh /tmp/test.txt
-rw-rw-r-- 1 kali kali 0 Mar  8 06:40 /tmp/test.txt
$ ./fchmodat2 /tmp/test.txt 777
$ ls -lh /tmp/test.txt
-rwxrwxrwx 1 kali kali 0 Mar  8 06:40 /tmp/test.txt
```

The rule is not triggered, no `SYSCALL` type alert is raised and the change to the file is unnoticed by the kernel:

```
# ausearch -k test_change_attr | grep fchmodat2
```

An attacker could use this to change file permissions in a stealthy way, given he has the right to do so in the first place.

This behaviour has been patched in kernel v7.0[7] and has been backported in the following LTS versions:

- 5.10
- 5.15
- 6.1
- 6.6
- 6.12

- 6.18
- 6.19

getxattrat and listxattrat

Extended attributes are key value pairs associated with files or directories[8]. These were at least already present in kernel v2.6.12. Several system calls defined by the patches allows one to deal with extended attributes, according to their respective man pages:

- `getxattr`: retrieves the value of the extended attribute identified by *name* and associated with the given *path* in the filesystem.
- `setxattr`: sets the *value* of the extended attribute identified by *name* and associated with the given *path* in the filesystem.
- `listxattr`: retrieves the list of extended attribute names associated with the given *path* in the filesystem.
- `removexattr`: removes the extended attribute identified by *name* and associated with the given *path* in the filesystem.

A set of userland tools also exists to manipulate extended attributes, namely `getfattr` and `setfattr`. Below is an example of adding an extended attribute (a checksum) to a file using these tools:

```
$ echo test > /tmp/test.txt
$ sha256sum /tmp/test.txt
f2ca1bb6c7e907d06dafe4687e579fce76b37e4e93b7605022da52e6ccc26fd2 /tmp/test.txt
$ setfattr --name=user.checksum --value="f2ca1bb6c7e907d06dafe4687e579fce76b37e4e93b760502
$ getfattr --encoding=text --dump /tmp/test.txt
# file: /tmp/test.txt
user.checksum="f2ca1bb6c7e907d06dafe4687e579fce76b37e4e93b7605022da52e6ccc26fd2"
```

The extended attributes are used for diverse purposes, amongst others SELinux uses it to store security labels[9] and the `security.capability` attribute is used to store file capabilities[10].

In kernel v6.13, the `*at` counterparts of the above system calls were added[11] but `getxattrat` and `listxattrat` were not added to the audit `read_class`.

As for the previously seen `fchmodat` and `fchmodat2` system calls, this can be proved by using the following two programs. The first using `getxattr`:

```c
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/xattr.h>

# define BUF_SIZE 4096

int main(int argc, char *argv[])
{
    char *path = argv[1];
    char *name = argv[2];
    char buf[BUF_SIZE];

    int status = getxattr(path, name, buf, BUF_SIZE);

    if (status == -1)
    {
        printf("getxattr failed: %s\n", strerror(errno));
    }

    buf[status] = 0;
    printf("%s: %s\n", name, buf);

    return 0;
}
```

And the second using `gexattrat`:

```c
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <linux/xattr.h>
#include <sys/syscall.h>
#include <unistd.h>

# define BUF_SIZE 4096

int main(int argc, char *argv[])
{
    char *path = argv[1];
    char *name = argv[2];
    char buf[BUF_SIZE];
```

```
    /* from include/uapi/linux/xattr.h
     struct xattr_args {
    __aligned_u64 __user value;
    __u32 size;
    __u32 flags;
    };
     */
    struct xattr_args args;
    args.value = (__u64)buf;
    args.size = BUF_SIZE;
    args.flags = 0;

    /*
     * from include/linux/syscalls.h
    asmlinkage long sys_getxattrat(int dfd, const char __user *path, unsigned int at_flags
                   const char __user *name,
                   struct xattr_args __user *args, size_t size);
    */
    int status = syscall(SYS_getxattrat, 0, path, 0, name, &args, sizeof(struct xattr_args

    if (status == -1)
    {
        printf("getxattrat failed: %s\n", strerror(errno));
    }

    buf[status] = 0;
    printf("%s: %s\n", name, buf);

    return 0;
}
```

And this time reusing the rule defined at the beginning of this article in `/etc/audit/rules.d/missing.rules`, as such:

```
-w /tmp/test.txt -p r -k test_read
```

When the `getxattr` program is executed against `/tmp/test.txt`:

```
$ ./getxattr /tmp/test.txt user.checksum
user.checksum: f2ca1bb6c7e907d06dafe4687e579fce76b37e4e93b7605022da52e6ccc26fd2
```

The `auditd` rule is triggered with a `SYSCALL` alert:

```
# ausearch -k test_read | grep getxattr
type=SYSCALL msg=audit(1772985246.140:640): arch=c000003e syscall=191 success=yes exit=64
```

While when the `getxattrat` program is used:

```
$ ./getxattrat /tmp/test.txt user.checksum
user.checksum: f2ca1bb6c7e907d06dafe4687e579fce76b37e4e93b7605022da52e6ccc26fd2
```

The rule does not get triggered:

```
# ausearch -k test_read | grep getxattrat
```

This allow for stealthy reconnaissance on a system, for example to query capabilities of files without the audit subsystem knowing about it:

```
$ cp $(which ping) /tmp
# setcap /tmp/ping cap_net_raw+eip
$ strace ./getxattrat /tmp/ping security.capability
...
getxattrat("/tmp/ping", "security.capability", "\1\0\0\2\0 \0\0\0 \0\0\0\0\0\0\0\0\0", 409
```

This behaviour was also patched in kernel v7.0[12] and backported in the same versions than the `fchmodat2` patch.

Conclusion

These flows are now fixed and I encourage you to update your kernel version.

These patches on missing system calls can make one wonder on how we can improve on detecting oversights in security features of the kernel. It is unclear who's role it is for example to add a system call to one of the audit subsystem class when such system call is been added to the kernel interface. Coordination between maintainers and patch authors may be the solution but error is human and one may forget to alert the audit subsystem maintainers, SELinux ones or any security-related feature ones. Maintainers, on the other side, have a lot to deal with and may not be able to keep up with every system calls being added.

References

1. Steve Grubb - audit.rules(7) - Linux man page ↩
2. Linus Torvalds - Linux kernel source tree ↩
3. Linux kernel maintainers - Adding a New System Call ↩
4. Ulrich Drepper - prototypes for *at functions & typo fix ↩
5. Alexey Gladkov, Palmer Dabbelt - fs: Add fchmodat2() ↩
6. Al Viro - audit syscall classes ↩
7. Jeffrey Bencteux - audit: add fchmodat2() to change attributes class ↩
8. man-pages project - xattr - Extended attributes ↩
9. RHEL - RHEL deployment guide - 49.4.3 SELinux Security Contexts ↩
10. man-pages project - capabilities(7) — Linux manual page ↩
11. Christian Göttsche - fs/xattr: add *at family syscalls ↩
12. Jeffrey Bencteux - audit: add missing syscalls to read class ↩