How Linux Capability Works in 2.6.25

1 Overview

The UNIX-style user privileges come in two varieties, `regular user` and `root`. Regular users' power is quite limited, while the `root` users are very powerful. If a process needs more power than those of regular users, the process is often running with the root privilege. Unfortunately, most of the time the processes do not actually need all the privileges. In other words, they have more powerful than what they need. This can pose serious risk when a process gets compromised. Therefore, having only two types of privileges is not sufficient; a more granular privilege set is required. The `POSIX` capabilities is exactly designed for this purpose.

2 How Linux Capability Works

2.1 Process Capability

Each Linux process has four sets of bitmaps called the *effective (E)*, *permitted (P)*, *inheritable (I)*, and *bset* capabilities. Each capability is implemented as a bit in each of these bitmaps, which is either `set` or `unset`.

```
struct task_struct
{
  kernel_cap_t cap_effective, cap_inheritable, cap_permitted, cap_bset;
}

typedef struct kernel_cap_struct {
    __u32 cap[_KERNEL_CAPABILITY_U32S];
} kernel_cap_t;
```

The constant `_KERNEL_CAPABILITY_U32S` indicates how many capabilities the kernel has, it would be defined to be 2 if kernel has more than 32 capabilities, otherwise, 1.

The *effective* capability set indicates what capabilities are effective. When a process tries to do a privileged operation, the operating system will check the appropriate bit in the effective set of the process (instead of checking whether the effective uid of the process i 0 as is normally done). For example, when a process tries to set the clock, the Linux kernel will check that the process has the `CAP_SYS_TIME` bit (which is currently bit 25) set in its effective set.

说明了effective和permitted的区别

The *permitted* capability set indicates what capabilities the process can use. The process can have capabilities set in the permitted set that are not in the effective set. This indicates that the process has temporarily disabled this capability. A process is allowed to set a bit in its effective set only if it is available in the permitted set. The distinction between effective and permitted makes it possible for a process to disable, enable and drop privileges.

The *inheritable* capability set indicates what capabilities of the current process should be inherited by the program executed by the current process. When a process executes a new program (using `exec()`), its new capability sets are calculated according to the following formula:

```
pI_new = pI
pP_new = (X & fP) | (fI & pI)
```

```
pE_new = pP_new   if fE == true
pE_new = empty    if fE == false
```

A value ending with 11new” indicates the newly calculated value. A value beginning with a `p` indicates a process capability. A value beginning with an `f` indicates a file capability. `X` indicates capability bounding set. This work is done by `cap_bprm_apply_creds()` in linux/security/commoncap.c.

Nothing special happens during `fork()` or `clone()`. Child processes and threads are given an exact copy of the capabilities of the parent process.

fork和clone的capability是不变的

The capability bounding set (`cap_bset`) is a set beyond which capabilities cannot grow. Previous kernels implement `cap_bset` for whole OS. You can find it in /proc/sys/kernel/cap-bound. Now each process has its own bounding set, which can be modified (droping only) via `prctl()`.

稍后需要了解bounding set

2.2 Manipulate Process Capability

Two system calls are provided to let users interact with process capabilities. They are `capget()` and `capset()` in kernel/capability.c. But unforturnately, with file capability support, process can only manipulate its own capability, this restriction is implemented in the following:

```
security/commoncap.c:
#ifdef CONFIG_SECURITY_FILE_CAPABILITIES
static inline int cap_block_setpcap(struct task_struct *target)
{
    /*
    * No support for remote process capability manipulation with
    * filesystem capability support.
    */
    return (target != current);
}
```

2.3 File Capability

To reduce the risk caused by `Set-UID` programs, we can assign a minimal set of capabilities to a privileged program, instead of giving the program the `root` privilege. Binding a set of capabilities to programs has been implemented since kernel 2.6.24. It is called *file capability*.

The basic idea is to assign certain attribute to the inode. Going through the process of `exec()` can give us a picture of how file capability works. (The capability-unrelated parts are omitted here)

```
in fs/exec.c:
int do_execve(...)
{
    prepare_binprm(bprm);
    search_binary_handler(bprm, regs);
}
```

Basically `prepare_binprm()` is to get capability from the inode. The function `search_binary_handler()` calls specific loading function of certain type of binary file, which finally calls `cap_bprm_apply_creds()` in the capability module. Its job is to apply the capability to the current process.

```
int prepare_binprm(struct linux_binprm *bprm)
{
    security_bprm_set(bprm);
}

in security/security.c:
int security_bprm_set(struct linux_binprm *bprm)
{
    return security_ops->bprm_set_security(bprm);
}
```

The `security_ops` points to secondary LSM. In 2.6.25, by default, it is capability module, which is stacked on SELinux module. Capability module is implemneted in security/commoncap.c. Since this module is always considered to be stacked on other modules, the hook functions in the module only do capability-related works, which do not cover all function points in struct security_operations (please refer to details on LSM mechanism). Here, `bprm_set_security()` points to `cap_bprm_set_security()`.

```
in security/commoncap.c:
int cap_bprm_set_security (struct linux_binprm *bprm)
{
    get_file_caps(bprm);
    if (!issecure (SECURE_NOROOT)) {
            if (bprm->e_uid == 0 || current->uid == 0) {
                    cap_set_full (bprm->cap_inheritable);
                    cap_set_full (bprm->cap_permitted);
            }
            if (bprm->e_uid == 0)
                    bprm->cap_effective = true;
    }
}
```

The function `get_file_caps(bprm)` first fetches the capability from the inode to struct linux_binprm. Then turn on all the capabilities if current user is root and SECURE_NOROOT is not set. SECURE_NOROOT is a security mode. SECURE_NO_SETUID_FIXUP is another one, when it is not set, then when a process switches its real or effective uids to or from 0, capability sets are further shifted around. 2.6.26 has more of them. We won't talk furture on this here.(check include/linux/securebits.h for the detailed definition)

2.4 Manipulating File Capability

Linux does not provide specific system call to manipulate file capability. But since it is implemented as inode attribute, we can use system call `getxattr()` and `fsetxattr()`. Please refer to `cap_get_file()` and `cap_set_file()` in `cap_file.c` in `libcap` for details on how to use it.

2.5 Checking Capability

The capabilities of a process are checked almost everywhere when an access attempt is made. Some of them can still grant permission even if ACL check fails. For example:

```
in fs/namei.c:
```

```
int generic_permission(...)
{
check_capabilities:
    /*
     * Read/write DACs are always overridable.
     * Executable DACs are overridable if at least one exec bit is set.
     */
    if (!(mask & MAY_EXEC) ||
        (inode->i_mode & S_IXUGO) || S_ISDIR(inode->i_mode))
            if (capable(CAP_DAC_OVERRIDE))
                return 0;
}
```

The function `capable(CAP_DAC_OVERRIDE)` checks whether the current process has `CAP_DAC_OVERRIDE` as an effective capability. The `capable()` function is linked to SELinux module function which is again linked to `cap_capable()` in the capability module as a seccondary module.

```
in security/commoncap.c:
int cap_capable (struct task_struct *tsk, int cap)
{
    /* Derived from include/linux/sched.h:capable. */
    if (cap_raised(tsk->cap_effective, cap))
        return 0;
    return -EPERM;
}
```

References

[1] Taking Advantage of Linux Capabilities. Available at
http://www.linuxjournal.com/article/5737

[2] Linux kernel capabilities FAQ. Available at
http://ftp.kernel.org/pub/linux/libs/security/linux-privs/kernel-2.4/capfaq-0.2.txt

[3] Linux Capabilities: making them work. Available at
http://ols.fedoraproject.org/OLS/Reprints-2008/hallyn-reprint.pdf

[4] POSIX file capabilities: Parceling the power of root. Available at
http://www.ibm.com/developerworks/library/l-posixcap.html?ca=dgr-lnxw01POSIX-capabilities