```
bob@alice:$ cat /tmp/test
TEST
```

In this way, operations can be blocked for specific groups.

Problem 4.12 Why is not even *root* allowed to execute a file if the execute bit is not set, although *root* has all permissions by definition? How is this related to the environment variable `PATH`, and why should "." not be added to the path?

Problem 4.13 Following up on the last problem, note that some people append "." as the last entry in the path and argue that only the first appearance of the executable file is used when typing a command. Is this variant also prone to potential security problems? If so, what are these problems?

4.3.2 Setuid and Setgid

In order to understand the set user ID and set group ID mechanisms, we take a closer look at how IDs are handled in Linux. Within the kernel, user names and group names are represented by unique non-negative numbers. These numbers are called the *user ID* (uid) and the *group ID* (gid), and they are mapped to the corresponding users and groups by the files `/etc/passwd` and `/etc/group`, respectively. By convention, the user ID `0` and the group ID `0` are associated with the superuser *root* and his user private group (UPG). User private groups is a concept where every system user is assigned to a dedicated group with the user's name. Note that some Linux distributions do not use UPGs but instead assign new users to a system-wide default group such as *staff* or *users*.

The Linux kernel assigns to every process a set of IDs. We distinguish between the *effective*, *real* and *saved* user ID (the situation is analogous for group IDs and we omit their discussion). The real user ID coincides with the user ID of the creator of the respective process. Instead of the real user ID, the effective user ID is used to verify the permissions of the process when executing system calls such as accessing files. Usually, real and effective IDs are identical, but in some situations they differ. This is used to raise the privileges of a process temporarily. To do this it is possible to set the *setuid* bit of an executable binary using the command `chmod u+s <executableBinary>`. If the setuid bit is set, the saved user ID of the process is set to the owner of the executable binary, otherwise to the real user ID. If a user (or more specifically a user's process) executes the program, the effective user ID of the corresponding process can be set to the saved user ID. The real user ID still refers to the user who invoked the program.

Example 4.1. The command `passwd` allows a user to change his password and therefore to modify the protected system file `/etc/passwd`. In order to allow ordinary users to access and modify this file without making it world accessible, the setuid bit of the binary `/usr/bin/passwd` is set. Since this binary is owned by *`root`*, the saved user ID of the respective process is set to `0` and the effective user ID is therefore also set to `0` whenever a user executes `passwd`. Thus, access to the `/etc/passwd` file is permitted. Since the real user ID still refers to the user, the program can determine which password in the file the user may alter.

For example, if user *`bob`* with user ID `17` executes the command `passwd`, the new process is first assigned the following user IDs:

real user ID:	`17`
saved user ID:	`0`
effective user ID:	`17`

The program passwd then invokes the system call `seteuid(0)` to set the effective user ID to `0`. Because the saved user ID is indeed `0`, the kernel accepts the call and sets the effective user ID accordingly:

real user ID:	`17`
saved user ID:	`0`
effective user ID:	`0`

Commands like `su`, `sudo` and many others also employ the setuid concept and allow an ordinary user to execute specific commands as *`root`* or any other user.

Setuid programs are often classified as potential security risks because they enable *privilege escalation*. If the setuid bit of a program owned by *`root`* is set, then an exploit can be used by an ordinary user to run commands as *`root`*. This might be done by exploiting a vulnerability in the implementation, such as a buffer overflow. However, when used properly, setuid programs can actually reduce security risks.

Note that for security reasons, the setuid bit of shell scripts is ignored by the kernel on most Linux systems. One reason is a race condition inherent to the way *shebang* (`#!`) is usually implemented. When a shell script is executed, the kernel opens the executable script that starts with a shebang. Next, after reading the shebang, the kernel closes the script and executes the corresponding interpreter defined after the shebang, with the path to the script added to the argument list. Now imagine that setuid scripts were allowed. Then an adversary could first create a symbolic link to an existing setuid script, execute it, and change the link right after the kernel opened the setuid script but before the interpreter opens it.

Example 4.2. Assume an existing setuid script `/bin/mySuidScript.sh` that is owned by *`root`* and starts with `#!/bin/bash`. An adversary could now proceed as follows to run his own `evilScript.sh` with *`root`* privileges:

```
mallet@alice:$ cd /tmp
mallet@alice:$ ln /bin/mySuidScript.sh temp
mallet@alice:$ nice -20 temp &
mallet@alice:$ mv evilScript.sh temp
```

The kernel interprets the third command as `nice -20 /bin/bash temp`. Since `nice -20` alters the scheduling priority to the lowest possible, the fourth command is likely to be executed before the interpreter opens `temp`. Hence, `evilScript.sh` gets executed with *`root`* permissions.

Problem 4.14 In what respect does the `passwd` command improve a system's security? How could an ordinary user be given the ability to change his own password without a setuid program?

Problem 4.15 Search for four setuid programs on **`alice`** and explain why they are setuid.

Note that instead of setuid, setgid can be used in many cases. Setgid is generally less dangerous than setuid. The reason is that groups typically have fewer permissions than owners. For example, a group cannot be assigned the permission to change access permissions. You can set a program's setgid bit using the command `chmod g+s <executableBinary>`.

Problem 4.16 Find some setgid programs on **`alice`** and explain why they are setgid. Could the setuid programs in the last problem be setgid instead of setuid?

4.4 Shell Script Security

Shell scripts offer a fast and convenient way to write simple programs and to perform repetitive tasks. However, as with programs written in other programming languages, shell scripts may contain security vulnerabilities. Therefore shell scripts must be designed, implemented and documented carefully, taking into account secure programming techniques as well as proper error handling.

We already discussed security flaws related to file system permissions. Furthermore, we pointed out that on modern Linux systems the kernel ignores the setuid and setgid bit of shell scripts by default. Therefore, some administrators run their shell scripts with *`root`* permissions to ensure that all commands used in the script have the permissions they require. In doing so, they give the script too many permissions, thereby contradicting the principle of least privilege. Whenever creating