Unix

G. Lettieri

21 September 2023

1 Introduction

During the lectures we will mostly use Linux. Linux is based on Unix, which is a completely different thing. It is important to set the record straight.

Unix was developed at AT&T Bell Labs in the late '60s by Ken Thompson, soon joined by Dennis Ritchie. It was a personal project that Thompson started on a PDP-7, when Bell Labs pulled out of the MULTICS project. It was then ported to a PDP-11 and enjoyed great success, first within the Bell Labs themselves and then at many universities and companies. The versions of Unix produced by Bell Labs are now called "Research Unix" and are numbered according to the version of their manual, starting with V1. The last well know Research Unix version is V7 from 1979; after that, and until the late '1990s, most people used a version of Unix that derived from either System V, a commercial version of Unix developed by AT&T, or BSD (Berkeley Software Distribution), a free version of Unix maintained by people at the University of California Berkeley (UC Berkeley). Both were based on research Unix code, extended in incompatible ways. The POSIX standard is (one of several) standards that try to define what every Unix system should look like.

A lawsuit was started in 1992 by AT&T against UC Berkeley, claiming that the latter had no right to use Unix code. Meanwhile, two important things happened: Richard Stallman had started the GNU (GNU's Not Unix) project in 1983, with the goal of rewriting Unix from the ground up to create a Unix-like system free of all legal problems. The GNU project produced many of the userspace commands, starting with the C compiler, but it never managed to materialize the kernel. This very important missing piece was finally created when Linus Torvalds announced his personal kernel project, which eventually became Linux, in 1991. By putting together Linux, the GNU software and a lot of other open source software (such as the Vim editor and the X Window System), we can now have several complete and free Unix-like systems.

The BSD people finally succeded in rewriting their software and freeing it from any dependence on Unix. The BSD system survives today in FreeBSD, OpenBSD, NetBSD and some other variations.

Bell Labs did not stop at V7 Unix, producing other less well-know versions up to V10, and developing "Plan 9 from Bell Labs" (a reference to the trash

sci-fi movie *Plan 9 from Outer Space*) and "Inferno", where the Unix ideas were developed without the need to maintain compatibility with ancient decisions. Today, Linux also includes ideas from Plan 9, such as the `clone()` primitive, which replaces the old `fork()`.

2 The structure of Unix

You have probabily encountered Unix in other courses, so we will move very quickly. For our purposes, one of the most important features to keep in mind is that no program in Unix is magic, except the kernel. Only the kernel can do things that the other programs cannot. In particular, the shell is not magic: everything the shell does, you can do in your own programs, and you can even replace the default shell with another one. Not even `sudo` is magic: it only allows you to become root temporarily thanks to the set-uid feature. Moreover, this feature is not specific to `sudo`: any executable can have the set-uid bit set at the discretion of its owner (or the administrator).

The kernel implements a number of primitives. Programs such as the shell, `ls`, `cat`, `sudo`, and so on, perform their tasks using these primitives. When you reason about what is possible and what is not, you only have to think about the primitives, not the programs: to read a file, `cat` must `open()` it, like any other program that reads a file.

The original Unix was admired for its elegance, since the set of primitives was very small (a modern Linux system, unfortunately, implements hundreds of primitives). The system implements processes, which are the entities that execute programs, and files, which are automatically expanded sequences of bytes organized in a hierarchy of pathnames (the file system). Processes are handled by the `fork()`, `execve()`, `exit()` and `wait()` primitives. Files are handled with the `open()`, `read()`, `write()` and `close()` primitives.

The kernel keeps track of a number of properties for each process. Among them are:

- the process identifier (a small number which is reused when a process terminates);
- a *real* and an *effective* user identifier, and a real and effective group identifier[1];
- a *current working directory*
- a table of open files.

The kernel also remembers, stored in the file system *inodes*, a set of attributes of each file and directory. Among them are:

- the file type (file, directory, symbolic link, device node)

[1]Later Unix-derived systems and modern Linux also implement a set of additional group identifiers, so a process can belong to many groups.

- the ids of the file's owner and group;
- the permission bits (read, write, execute permissions for the file's owner, for the file's group, and for other users)
- the set-uid and set-gid flags.

2.1 Process primitives

The `fork()` primitive is the only way to create a new process. The child process is a copy of its parent and, in particular, all of the process attributes mentioned above are copied to the child process (except, of course, the process identifier). In particular, the new process will have the same uid and gid as its parent, and will share the same open files. The executed program is also the same: the `fork()` primitive will return different values to the parent and the child, so the program can take different branches and let the two processes perform different actions.

The `execve()` primitive is the only way to execute a new program. The first argument of the primitive is the path of a file, which must be executable by the calling process (according to the file's permissions and the process's *effective* uid and gid). The process remains the same: same id, same open files, same current working directory, same real uid and gid (the effective ones may change because of the set-uid and set-gid feature, see below). However, all its memory is replaced by the contents of the file. Execution resumes at the file's `_start` symbol. The `execve()` primitive takes two more arguments, which are two arrays of C strings, with each array terminated by `NULL`. The code in `_start` (which comes from the standard C library) will use the first array to create the `argc` and `argv` parameters of `main`, while the second array is used as the set of the *environment variables* of the process, pointed to by the global `environ` variable.

If the set-uid bit of the file is set, `execve()` changes the effective uid of the process to the uid of the owner of the file. The same goes for the set-gid bit and the effective gid. The real uid and gid do not change.

Processes terminate when they call the `exit()` primitive. A process can wait for any of its children to exit by using the `wait()` primitive. A small integer passed to `exit()`, can be received by `wait()` and used as a means for the child process to communicate errors or special conditions to its parent. By convention, a value of 0 means that all was well.

The uid and gid of a process can also be changed using the `setuid()` and `setgid()` primitives which behave differently when called by the root user (uid 0) and normal users (uid different from 0). If the effective uid of the calling process is 0, `setuid()` will accept any value and set both the real and effective uids to it (and similarly for `setgid()`). Normal users can only call these primitives when they are essentially no-ops (setting the ids to the value they already have).

2.2 File primitives

We will cover only the most important features, assuming that you are already familiar with a Unix-like file system.

Files must be opened before they can be used. The `open()` primitive takes a path as the first argument and a mode (read and/or write) as the second. It checks if the calling process has the proper permissions to open the file in that mode (again, using the effective uid/gid and the file permissions). If successful, it finds the first free slot in the process open-file table, stores there a pointer to an in-kernel data structure describing the open file, and returns the index of the slot. This index (file descriptor) can be passed to the `read()` and `write()` system calls to sequentially read and write bytes to and from the open file. The process should `close()` on the files when it is done with them, mostly because the open-file table has a finite size, but the kernel will automatically close any open files when the process terminates.

If so instructed, `open()` can also create the file if it does not exist. The original design used a different primitive for this, `creat()`, which is still available today.

In the original Unix, directories were accessed using the same primitives as above, but more specialized primitives were added later (`opendir()`, `readdir()`, `closedir()`).

Finally, a process can change its own current working directory using the `chdir()` primitive, passing the path of the new directory as an argument.

2.3 Interpreting paths

All of `execve()`, `open()` and `chdir()` always interpret their first argument—a filesystem path—in the same way:

- if it *begins* with `/`, it is an absolute path starting from the root of the file system;
- if it *does not begin* with `/`, it is a relative path starting in the current working directory of the process.

Note, in particular, that no `PATH` variable is taken into account by `execve()`: `PATH` is handled in userspace before calling `execve()`, as we will see.

Any character can be part of a file or directory name, except 0 and `/`. This includes whitespace, even newlines, and control characters like backspace.

2.4 How things fit together

From a Unix access control point of view, you must always remember that what matters is the effective uid and gid of the processes, as checked by the kernel when they invoke primitives. The programs are not important: exactly the same `ls` program is run by normal users and by root, but the set of directories that can be listed depends on *who* is running `ls`. More specifically, it depends on the effective uid and gid of the process that `execve()`'d the `/bin/ls` file.

So, how do processes get the effective uids and gids of users? When Unix boots, the kernel creates a process with id 1, both real and effective uid and gid set to 0 and no open files. The process executes the `init` command, which must exist and be executable. Consider the original Unix setup, which is simpler: the Unix system is running on a large computer somewhere, and there are several terminals connected to it. The available terminals are listed, one per line, in a `/etc/ttys` file created by the administrator (ttys stands for teletypes). For each teletype mentioned in the file, `init` forks and executes the `getty` command, passing the name of the teletype as an argument (via the second parameter of `execve()`). It then waits endlessly for any of its children to exit. Whenever a child exits, `init` wakes up and spawns a new `getty` for the corresponding terminal.

The `getty` process thus starts with uid and gid set to zero and no file open. It does whatever is necessary to put the teletype device into a usable state, then it opens the corresponding device file *three times*, once in read-only mode and twice in write-only mode. Since there were no open files before, and the `open()` primitive always uses the first free slot in the file descriptor table, the device will go into file descriptors 0, 1 and 2. From this point on, all other programs that are run, and processes created directly or indirectly from this process, will start with these file descriptors already open. By convention, these are the standard input, standard output and standard error files for programs started from this terminal. The `getty` program will then print "`login:`" to file descriptor 1 and start reading from file descriptor 0. When a user comes to the terminal and enters her name, `getty` will `execve()` the `login` program, passing the entered username as an argument.

The `login` process still has its uids and gids set to zero, but the fds 0, 1 and 2 now point to the terminal. It prints "`password:`" to file descriptor 1, then reads from file descriptor 0 (it has to issue some `ioctl()` first, so that the terminal does not echo the password to the terminal printer, where it would remain there for everyone to see). Once it gets the password, it opens and reads the `/etc/passwd` file (prepared by the administrator), looking for a line starting with the username. If it finds it, it checks the password, and if they match, it reads the `uid`, `gid`, `home_directory` and `shell` from the rest of the line, then it calls

- `setgid(gid);`
- `setuid(uid);`
- `chdir(working_directory);`
- `execve(shell, ...);`

Since `login` is still running as root, the first two calls set both the real and effective gids and uids of the process those of the logged-in user. When the shell runs, it will still have file descriptors 0, 1 and 2 pointing to the terminal opened by `getty` and the uids and gids set by `login`. From now on, all the programs started by the shell will inherit this setting, and there will be no

way to arbitrarily change the uid and gid (unless they are still 0 because the logged-in user is actually the administrator), since `setuid()` and `setgid()` are no longer available. Only by `execve()`ing set-uid/set-gid programs can the uids and gids be changed, but these programs will only perform safe actions (or so the administrator hopes).

When the shell exits because the user logs out, `init` will wake up and respawn `getty` on the terminal (the shell is still running in the child process initially forked from `init`, since only `execve()` has been called since then).

2.5 Is Unix secure?

No, according to Dennis Ritchie himself. You can read what he had to say on the subject here:

`http://www.tom-yam.or.jp/2238/ref/secur.pdf`

Unfortunately, many of the things he said are still true today. Some have gotten better, but some have gotten worse. The root user, for example, has become more and more powerful over the years. Contrast this with Plan 9, where the root user was completely eliminated.

Finding ways to undermine Unix (Linux, actually) security is what we will try to do in the next few lectures. The ultimate goal of an attacker will be to obtain a *root shell*, i.e., a process with effective uid 0 running a shell. From there, the attacker can do whatever she wants, including hiding her own tracks.