Unix

Federico Casu

February 28, 2024

The structure of Unix

The **System and Network Hacking** course primarily focuses on the `Linux` operating system. `Linux` is built upon the foundations of `Unix`; therefore, it is essential to delve into `Unix` OS to gain a deeper understanding of what is under the hood of `Linux`.

First thing first: no program in `Unix` is magic. Any action that a program can perform is granted, somehow, by the *kernel*. The kernel is responsible for building a layer of indirection between the programs and the hardware. A program can carry out its tasks by requesting help from the kernel, which is provided in the form of *primitive calls*.

For example, let's consider the `cat` command:

1. `cat` needs to read the content from the file specified by the user.
2. Additionally, the content of the file needs to be printed, in `ASCII` format, on the display.

What primitives does the kernel need to make available to `cat`?

1. The file is likely stored persistently somewhere on a HDD. Thus, the kernel should provide a primitive capable of reading the content from the target file.
2. To display the content of the file in a human readable format, the kernel should provide a primitive that enables interaction with the display controller.

Here it is: `Unix` is based on primitives. When you reason about what is possible and what is not, you only have to think about the primitives, not the programs.

How can `cat` be implemented? Let's find out:

Listing 1: `cat.c`

```c
int _syscall(uint64_t, uint64_t, uint64_t, uint64_t);
int _stdin__stdout();

int main(int argc, char* argv[]) {
    // No arguments: no file to open.
    // cat will copy stdin to stdout.
    if (argc == 1) {
        int res = _stdin__stdout();
        ERROR(res);
        _syscall(res, 0, 0, SYS_EXIT);  // exit(res);
    }

    int fd;
    char tmp;

    // We start opening files from argv[1] because
    // argv[0] == 'cat'.
    for (int i = 1; i < argc; i++) {
        fd = _syscall((uint64_t) argv[i], O_RDONLY, 0, SYS_OPEN);

        FILE_ERROR(fd, argv[i]);

        while (_syscall(fd, &tmp, sizeof(char), SYS_READ) != 0)
            _syscall(STDOUT, &tmp, sizeof(char), SYS_WRITE);

        FILE_ERROR(_syscall(fd, 0, 0, SYS_CLOSE), argv[i]);
    }

    _syscall(0, 0, 0, SYS_EXIT);    // exit(0);
}
```

Listing 2: `syscall` wrapper

```c
int _syscall(uint64_t arg1, uint64_t arg2, uint64_t arg3,
             uint64_t sys_number) {
    int result;
    asm volatile (
        "movq %1, %%rdi\n"   // Move arg1 to %rdi
        "movq %2, %%rsi\n"   // Move arg2 to %rsi
        "movq %3, %%rdx\n"   // Move arg3 to %rdx
        "movq %4, %%rax\n"   // Move sys_number to %rax
        "syscall\n"
        "movl %%eax, %0\n"   // Move return value to 'result'

        : "=r" (result)      // Output: result
        : "r" (arg1), "r" (arg2), "r" (arg3), "r" (sys_number)
        : "%rax", "%rdi", "%rsi", "%rdx", "cc", "memory"
    );
    return result;
}
```

Listing 3: Utility function to copy `stdin` to `stdout`

```c
int _stdin__stdout() {
    char tmp, buffer[BUFFER_SIZE];
    int  count = 0, n_read = 0;

    // Read from stdin one byte at time
    while ((n_read =
            _syscall(STDIN, (uint64_t) &tmp, 1, SYS_READ)) == 1) {
        buffer[count++] = tmp;

        // Check overflow
        if (!(count < BUFFER_SIZE - 1))
            buffer[count++] = tmp = '\n';

        if (tmp == '\n') {
            // Copy temporary buffer to stdout
            if (_syscall(STDOUT, (uint64_t) buffer, count,
    SYS_WRITE) == -1)
                return -1;
            // Flush temporary buffer
            count = 0;
        }
    }

    // n_read == -1: error
    // n_read == 0:  everything OK, user pressed Ctrl^D to exit
    return n_read;
}
```

Unix is a multiprogrammed operating system, i.e., the system implements processes. Such processes are meant to carry out the tasks the user has requested to be performed by executing a program. Therefore, `Unix` implements a set of primitives that can be used to spawn and manage processes.

```c
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

`fork()` creates a new process by duplicating the calling process. The child process and the parent process run in separate memory spaces. At the time of `fork()` both memory spaces have the same content. Memory writes, file mappings (`mmap()`), and unmappings (`munmap()`) performed by one of the processes do not affect the other.

```c
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *wstatus);
```

`wait()` is used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a zombie state.

```c
#include <unistd.h>

int execve(const char *pathname, char *const argv[],
           char *const envp[]);
```

`execve()` executes the program referred to by pathname. This causes the program that is currently being run by the calling process to be replaced with a new program, with newly initialized stack, heap, and (initialized and uninitialized) data segments. `pathname` must be either a binary executable, or a script starting with a line of the form:

```
#!interpreter [optional-arg]
```

`argv` is an array of pointers to strings passed to the new program as its command-line arguments. By convention, the first of these strings (i.e., `argv[0]`) should contain the filename associated with the file being executed. The `argv` array must be terminated by a `NULL` pointer (Thus, in the new program, `argv[argc]` will be `NULL`). `envp` is an array of pointers to strings, conventionally of the form `key=value`, which are passed as the environment of the new program. The `envp` array must be terminated by a `NULL` pointer. If the `set-user-ID` bit is set on the program file referred to by pathname, then the effective user ID of the calling process is changed to that of the owner of the program file. Similarly, if the `set-group-ID` bit is set on the program file, then the effective group ID of the calling process is set to the group of the program file.

The kernel maintains various information about each process running on the system. Among them, the ones we are interested in are the following:

1. Process ID (`PID`): A unique identifier assigned to each process running on the system.
2. Parent Process ID (`PPID`): The `PID` of the parent process that spawned the current process.
3. *Real* and *effective* User ID, *real* and *effective* Group ID: These are the user and group identifiers associated with the process, which determine the permissions and access rights of the process. If the `suid` (`sgid`) bit is not set, the real and effective IDs are the same. However, if the `suid` (`sgid`) bit is set, the effective user (group) ID is changed to the user (group) ID of the owner of that executable. This means that the executable will run

with the privileges of the file's owner rather than the privileges of the user executing it.

4. File Descriptors: A list of file descriptors associated with the process, including open files, pipes, sockets, and other communication endpoints.

5. Working Directory: The current working directory of the process, which determines the base directory for relative file paths.

From an access control perspective, what matters are the effective user and group identifiers of a process. How does a process get its own user and group identifiers?

- When `Unix` boots, the kernel creates a process with ID 1, setting both real and effective user IDs and group IDs to 0. This process executes the `init` command.

- The original `Unix` system runs on a mainframe with multiple terminals connected to it. Each available terminal is listed in `/etc/ttys` file. For each terminal mentioned in this file, the `init` process `fork()`s a new process and executes the `getty` command within the newly spawned process.

- Each `getty` process starts with both `uid` and `gid` set to 0 and no open files. It first prepares the terminal device for use and then `open()`s the corresponding device file three times: once in read-only mode and twice in write-only mode. As there were no open files previously, the device file descriptors are assigned slots 0, 1, and 2 in the file descriptor table. These file descriptors conventionally represent standard input (`stdin`), standard output (`stdout`), and standard error (`stderr`), respectively, for any program that will inherit such a terminal.

- After printing "`login:`", `getty` awaits user input from `stdin`. When a user interacts with the terminal by entering their name, `getty` invokes the `execve()` system call, passing the path of the `login` program and the username as arguments.

- The login process, still with `uid` and `gid` set to 0, now has file descriptors 0, 1, and 2 pointing to the terminal. It prints "`password:`" to file descriptor 1 and reads from file descriptor 0. After issuing some `ioctl()` commands to ensure password confidentiality, `login` reads the `/etc/passwd` file to verify the username and password. If successful, `login` retrieves the `uid`, `gid`, home directory, and shell from the `/etc/passwd` file and calls:

  ```
  setgid(gid);
  setuid(uid);
  chdir(work_dir);

  char *argv = {shell, NULL};
  char *envp = {NULL};

  execve(shell, argv, envp);
  ```

- These calls set both the real and effective `uid`s and `gid`s of the process to those of the logged-in user. When the shell runs, it inherits the file descriptors pointing to the terminal opened by `getty` and the user IDs set by login. Subsequently, all programs started by the shell inherit these settings.