1 The /proc File System

- In the old days, certain UNIX programs, such as **ps** and **uptime**, accessed kernel data structures directly to retrieve system information.
  - This required knowledge of kernel externals and required care to ensure that the values were not modified as they were being accessed.
  - The programs needed to be *setuid* root in order to access the kernel data structures; this meant they were vulnerable to security exploits if they were not carefully written.
  - These programs also frequently had to be rebuilt when the kernel was changed because the positions and layouts of the data structures may have changed.
- Modern systems implement a **/proc** filesystem that contains special files that can be read to access system status information.
- **/proc** is a window into the running Linux kernel. Files in the **/proc** file system don't correspond to actual files on a physical device.
  - Instead, they are magic objects that behave like files but provide access to parameters, data structures, and statistics in the kernel.
- The contents of these files are not always fixed blocks of data, as ordinary file contents are.
  - Instead, they are generated on the fly by the Linux kernel when you read from the file.
- You can also change the configuration of the running kernel by writing to certain files in the **/proc** file system.
- The code that implements the **/proc** filesystem and some of its entries can be found in */usr/src/linux/fs/proc*. Portions of the proc namespace are registered using the kernel function **proc_register()**.

```
$ ls -l /proc/version
$ cat /proc/version
$ man 5 proc
```

1.1 Extracting Information from /proc

- Most of the entries in /proc provide information formatted to be readable by humans.

  ```
  $ cat /proc/cpuinfo
  ```

- The program in Fig. 1 shows a simple way to extract a value from the output by reading the file into a buffer and parse it in memory using **sscanf**. The program includes the function **get_cpu_clock_speed** that reads from **/proc/cpuinfo** into memory and extracts the first CPU's clock speed.

1.2 Process Information

- The **/proc** file system contains a directory entry for each process running on the system.
- The name of each directory is the process ID of the corresponding process.
- Each process directory contains these entries:
  - **cmdline** contains the argument list for the process.
    - The command line may be blank for zombie processes and the arguments might not be available if the process is swapped out.
    - You can take a quick look at what is running on your system by issuing the following command:

      ```
      $ strings -f /proc/[0-9]*/cmdline
      ```
  - **cwd** is a symbolic link that points to the inode for the current working directory of the process.
  - **cpu** entry appears only on SMP Linux kernels. It contains a breakdown of process time (user and system) by CPU.
  - **environ** contains the process's environment.
  - **exe** is a symbolic link that points to the executable image running in the process. This could also point to a script that is being executed or the executable processing it, depending on the nature of the script.

```
#include <stdio.h>
#include <string.h>
/* Returns the clock speed of the system's CPU in MHz, as reported by
   /proc/cpuinfo.  On a multiprocessor machine, returns the speed of
   the first CPU.  On error returns zero.  */
float get_cpu_clock_speed ()
{
  FILE* fp;
  char buffer[1024];
  size_t bytes_read;
  char* match;
  float clock_speed;
  /* Read the entire contents of /proc/cpuinfo into the buffer.  */
  fp = fopen ("/proc/cpuinfo", "r");
  bytes_read = fread (buffer, 1, sizeof (buffer), fp);
  fclose (fp);
  /* Bail if read failed or if buffer isn't big enough.  */
  if (bytes_read == 0 || bytes_read == sizeof (buffer))
    return 0;
  /* NUL-terminate the text.  */
  buffer[bytes_read] = '\0';
  /* Locate the line that starts with "cpu MHz".  */
  match = strstr (buffer, "cpu MHz");
  if (match == NULL)
    return 0;
  /* Parse the line to extract the clock speed.  */
  sscanf (match, "cpu MHz : %f", &clock_speed);
  return clock_speed;
}
int main ()
{
  printf ("CPU clock speed: %4.0f MHz\n", get_cpu_clock_speed ());
  return 0;
}
```

Figure 1: Extract CPU Clock Speed from **/proc/cpuinfo**.

- **fd** is a subdirectory that contains entries for the file descriptors opened by the process as a symbolic link to the actual file's inode.
  - Opening the file descriptor entry (for example, **/proc/self/fd/1**) opens the file itself; if the file is a terminal or other special device, it may interfere with the process itself by stealing data.
- **maps** displays information about files mapped into the process's address. For each mapped file, **maps** displays the range of addresses in the process's address space into which the file is mapped, the permissions on these addresses, the name of the file, and other information.
  The **maps** table for each process displays the executable running in the process, any loaded shared libraries, and other files that the process has mapped in.
- **root** is a symbolic link to the root directory for this process. Usually, this is a symbolic link to /, the system root directory. (chroot)
- **stat** contains lots of status and statistical information about the process. These are the same data as presented in the **status** entry, but in raw numerical format, all on a single line.
- **statm** contains information about the memory used by the process.
  - The variable names, in order, used in array.c are *size, resident, share, trs, lrs, drs,* and *dt.*
  - These give the total size (including code, data, and stack), resident set size, shared pages, text pages, stack pages, and dirty pages.
- **status** contains lots of status and statistical information about the process, formatted to be comprehensible by humans. It includes the name, state, process id, parent process id, uids and group ids (including real, effective, and saved ids), virtual memory statistics, and signal masks.
- **mem** entry can be used to access the memory image of a particular process.

- The source file */usr/src/linux/fs/proc/array.c* seems to have most of the routines that actually generate **/proc** output for the per process entries.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
/* Returns the process id of the calling processes, as determined from
   the /proc/self symlink.  */
pid_t get_pid_from_proc_self ()
{
  char target[32];
  int pid;
  /* Read the target of the symbolic link.  */
  readlink ("/proc/self", target, sizeof (target));
  /* The target is a directory named for the process id.  */
  sscanf (target, "%d", &pid);
  return (pid_t) pid;
}
int main ()
{
  printf ("/proc/self reports process id %d\n",
  (int) get_pid_from_proc_self ());
  printf ("getpid() reports process id %d\n", (int) getpid ());
  return 0;
}
```

Figure 2: Obtain the Process ID from **/proc/self**.

1.2.1 /proc/self

- One additional entry in the **/proc** file system makes it easy for a program to use **/proc** to find information about its own process.
- The entry **/proc/self** is a symbolic link to the **/proc** directory corresponding to the current process.
- The destination of the **/proc/self** link depends on which process looks at it: Each process sees its own process directory as the target of the link.
- The program in Fig. 2 reads the target of the **/proc/self** link to determine its process ID.

1.2.2 Process Argument List

- The arguments are presented as a single character string, with arguments separated by **NUL**s.
- **NUL** vs. **NULL**. **NUL** is the character with integer value 0. It is different from **NULL**, which is a pointer with value 0.
- **NULL** is a pointer value that you can be sure will never correspond to a real memory address in your program.
- In C and C++, **NUL** is expressed as the character constant '\0', or (char) 0. The definition of **NULL** differs among operating systems; on Linux, it is defined as **((void*)0)** in C and simply **0** in C++.
- The program Fig. 3 prints the argument list of the process with the specified process ID.

```
$ ./print-arg-list PID
```

1.2.3 Process Environment

- The **environ** entry contains a process's environment.
- The program Fig. 4 takes a process ID number on its command line and prints the environment for that process by reading it from **/proc**.

1.2.4 Process Executable

- The **exe** entry points to the executable file being run in a process.
- Typically the program executable name is passed as the first element of the argument list.
- Using the **exe** entry in the **/proc** file system is a more reliable way to determine which executable is running.
- The function **get_executable_path** in the program (See Fig. 5) determines the path of the executable running in the calling process by examining the symbolic link **/proc/self/exe**.

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
/* Prints the argument list, one argument to a line, of the process
   given by PID.  */
void print_process_arg_list (pid_t pid)
{
  int fd;
  char filename[24];
  char arg_list[1024];
  size_t length;
  char* next_arg;
  /* Generate the name of the cmdline file for the process. */
  snprintf (filename, sizeof (filename), "/proc/%d/cmdline", (int) pid);
  /* Read the contents of the file. */
  fd = open (filename, O_RDONLY);
  length = read (fd, arg_list, sizeof (arg_list));
  close (fd);
  /* read does not NUL-terminate the buffer, so do it here. */
  arg_list[length] = '\0';
  /* Loop over arguments.  Arguments are separated by NULs. */
  next_arg = arg_list;
  while (next_arg < arg_list + length) {
    /* Print the argument.  Each is NUL-terminated, so just treat it
       like an ordinary string.  */
    printf ("%s\n", next_arg);
    /* Advance to the next argument.  Since each argument is
       NUL-terminated, strlen counts the length of the next argument,
       not the entire argument list.  */
    next_arg += strlen (next_arg) + 1;
  }
}
int main (int argc, char* argv[])
{
  pid_t pid = (pid_t) atoi (argv[1]);
  print_process_arg_list (pid);
  return 0;
}
```

Figure 3: Print the Argument List of a Running Process.

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
/* Prints the environment, one environment variable to a line, of the
   process given by PID.  */
void print_process_environment (pid_t pid)
{
  int fd;
  char filename[24];
  char environment[8192];
  size_t length;
  char* next_var;
  /* Generate the name of the environ file for the process. */
  snprintf (filename, sizeof (filename), "/proc/%d/environ", (int) pid);
  /* Read the contents of the file. */
  fd = open (filename, O_RDONLY);
  length = read (fd, environment, sizeof (environment));
  close (fd);
  /* read does not NUL-terminate the buffer, so do it here. */
  environment[length] = '\0';
  /* Loop over variables.  Variables are separated by NULs. */
  next_var = environment;
  while (next_var < environment + length) {
    /* Print the variable.  Each is NUL-terminated, so just treat it
       like an ordinary string.  */
    printf ("%s\n", next_var);
    /* Advance to the next variable.  Since each variable is
       NUL-terminated, strlen counts the length of the next variable,
       not the entire variable list.  */
    next_var += strlen (next_var) + 1;
  }
}
int main (int argc, char* argv[])
{
  pid_t pid = (pid_t) atoi (argv[1]);
  print_process_environment (pid);
  return 0;
}
```

Figure 4: Display the Environment of a Process.

```
#include <limits.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
/* Finds the path containing the currently-running program executable.
   The path is placed into BUFFER, which is of length LEN.  Returns
   the number of characters in the path, or -1 on error.  */
size_t get_executable_path (char* buffer, size_t len)
{
  char* path_end;
  /* Read the target of /proc/self/exe.  */
  if (readlink ("/proc/self/exe", buffer, len) <= 0)
    return -1;
  /* Find the last occurrence of a forward slash, the path separator.  */
  path_end = strrchr (buffer, '/');
  if (path_end == NULL)
    return -1;
  /* Advance to the character past the last slash.  */
  ++path_end;
  /* Obtain the directory containing the program by truncating the
     path after the last slash.  */
  *path_end = '\0';
  /* The length of the path is the number of characters up through the
     last slash.  */
  return (size_t) (path_end - buffer);
}
int main ()
{
  char path[PATH_MAX];
  get_executable_path (path, sizeof (path));
  printf ("this program is in the directory %s\n", path);
  return 0;
}
```

Figure 5: Get the Path of the Currently Running Program.

1.2.5 Process File Descriptors

- The **fd** entry is a subdirectory that contains entries for the file descriptors opened by a process. Each entry is a symbolic link to the file or device opened on that file descriptor.

- You can write to or read from these symbolic links; this writes to or reads from the corresponding file or device opened in the target process. The entries in the **fd** subdirectory are named by the file descriptor numbers.

- Open a new window, and find the process ID of the shell process by running **ps**.

  ```
  $ ps
  ```

- Now open a second window, and look at the contents of the fd subdirectory for that process.

  ```
  $ ls -l /proc/PID/fd
  ```

- File descriptors 0, 1, and 2 are initialized to standard input, output, and error, respectively.

- Thus, by writing to **/proc/PID/fd/1**, you can write to the device attached to stdout for the shell process.

- In the second window, try writing a message to that file:

  ```
  $echo "Hello, world." >> /proc/PID/fd/1
  ```

- The program in Fig. 6 presents a program that simply opens a file descriptor to a file specified on the command line and then loops forever.

  ```
  $ ./open-and-spin open-and-spin.c (in one window)
  $ ls -l /proc/PID/fd (in other window)
  ```

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int main (int argc, char* argv[])
{
  const char* const filename = argv[1];
  int fd = open (filename, O_RDONLY);
  printf ("in process %d, file descriptor %d is open to %s\n",
  (int) getpid (), (int) fd, filename);
  while (1);
  return 0;
}
```

Figure 6: Open a File for Reading.

1.2.6 Process Memory Statistics

- The **statm** entry contains a list of seven numbers, separated by spaces. Each number is a count of the number of pages of memory used by the process in a particular category. The categories, in the order the numbers appear, are listed here:

  1. The total process size.
  2. The size of the process resident in physical memory.
  3. The memory shared with other processes; that is, memory mapped both by this process and at least one other (such as shared libraries or untouched copy-on-write pages).
  4. The text size of the process; that is, the size of loaded executable code.
  5. The size of shared libraries mapped into this process.
  6. The memory used by this process for its stack.
  7. The number of dirty pages; that is, pages of memory that have been modified by the program.

1.3 Hardware Information

- Several of the other entries in the **/proc** file system provide access to information about the system hardware.

- **CPU Information; /proc/cpuinfo** contains information about the CPU or CPUs.
  - The *Processor* field lists the processor number; this is 0 for single-processor systems.
  - The *Vendor*, *CPU Family*, *Model*, and *Stepping* fields enable you to determine the exact model and revision of the CPU.
  - More useful, the *Flags* field shows which CPU flags are set, which indicates the features available in this CPU. For example, **mmx** indicates the availability of the extended MMX instructions.
  - The last element, *bogomips*, is a Linux-specific value. It is a measurement of the processor's speed spinning in a tight loop and is therefore a rather poor indicator of overall processor speed.
- **Device Information;** the **/proc/devices** file lists major device numbers for character and block devices available to the system.
- **PCI Bus Information;** the **/proc/pci** file lists a summary of devices attached to the PCI bus or buses. These are actual PCI expansion cards and may also include devices built into the system's motherboard, plus AGP graphics cards.
- **Serial Port Information;** the **/proc/tty/driver/serial** file lists configuration information and statistics about serial ports. For example, this line from **/proc/tty/driver/serial** might describe serial port 1 (which would be COM2 under Windows).
- **DMA Information;** the **/proc/dma** file lists which DMA channels have been reserved by drivers and the name the driver gave when reserving them. The cascade entry is for the DMA line that is used to cascade the secondary DMA controller off of the primary controller; this line is not available for other use.
- **Interrupt Information; /proc/interrupts** file has one line per reserved interrupt.
  - The fields are the interrupt number, the number of interrupts received on that line, a field that may have a plus sign (SA_INTERRUPT flag set) and the name a driver used when registering that interrupt.
  - The function **get_irq_list()** in */usr/src/linux/arch/i386/kernel/irq.c* (assuming Intel platform) generates this data.

  - This is a very handy file to manually "cat" before installing new hardware, as are **/proc/dma** and **/proc/ioports**. They list the resources that are currently in use (but not those used by hardware for which no driver is loaded).

- **IOPorts Information; /proc/ioports** file lists the various I/O port ranges registered by various device drivers such as your disk drives, ethernet, and sound devices.

- **Raid Devices;** textbf/proc/mdstat file contains information on raid devices controlled by the md device driver.

- **Memory Information; /proc/meminfo** file gives information on memory status and is used by the free program. Its format is similar to that displayed by **free**. This displays the total amount of free and used physical and swap memory in the system. This also shows the shared memory and buffers used by the kernel.

- **Clock Information; /proc/rtc** file gives information on the hardware real-time clock including the current date and time, alarm setting, battery status, and various features supported. The **/sbin/hwclock** command is normally used to manipulate the real-time clock.

- **Net Information; /proc/net** subdirectory contains files that describe and/or modify the behavior of the networking code. Many of these special files are set or queried through the use of the **arp, netstat, route,** and **ipfwadm** commands.

- **SCSI Information; /proc/scsi** subdirectory contains one file that lists all detected SCSI devices and one directory for each controller driver, with a further subdirectory for each separate instance of that controller installed.

1.3.1 Kernel Information

- Many of the entries in **/proc** provide access to information about the running kernel's configuration and state. Some of these entries are at the top level of **/proc**; others are under **/proc/sys/kernel**.

  - **kcore Information; /proc/kcore** file is the physical memory of the system in core file format. It is used with GDB to examine kernel data structures. This file is not in a text format.

- **kmsg Information; /proc/kmsg** file; only one process, which must have root privileges, can read this file at any given time. This is used to retrieve kernel messages generated using **printk()**.
- **ksyms Information; /proc/ksyms** file lists the kernel symbols that have been registered; these symbols give the address of a variable or function.
  * Each line gives the address of a symbol, the name of the symbol, and the module that registered the symbol.
  * The **ksyms, insmod,** and **kmod** programs probably use this file. It also lists the number of running tasks, the total number of tasks, and the last pid assigned.
- **Modules Information; /proc/modules** file gives information on loadable kernel modules. This information is used by the **lsmod** program to display the information on the name, size, usecount, and referring modules.
- **Version Information;** the file **/proc/version** contains a long string describing the kernel's release number and build version. It also includes information about how the kernel was built: the user who compiled it, the machine on which it was compiled, the date it was compiled, and the compiler release that was used.

  ```
  $ cat /proc/version
  $ cat /proc/sys/kernel/ostype
  $ cat /proc/sys/kernel/osrelease
  $ cat /proc/sys/kernel/version
  ```

- **Hostname and Domain Name;** the **/proc/sys/kernel/hostname** and **/proc/sys/kernel/domainname** entries contain the computer's hostname and domain name, respectively.
- **Memory Usage;** the /proc/meminfo entry contains information about the system's memory usage.

  ```
  $ cat /proc/meminfo
  ```

  * The *Shared* column displays total shared memory currently allocated on the system.
  * The *Buffers* column displays the memory allocated by Linux for block device buffers. These buffers are used by device drivers to hold blocks of data being read from and written to disk.

    * The *Cached* column displays the memory allocated by Linux to the page cache. This memory is used to cache accesses to mapped files.
    * You can use the **free** command to display the same memory information.

1.3.2 Drives, Mounts, and File Systems

- The **/proc** file system also contains information about the disk drives present in the system and the file systems mounted from them.
- **File Systems;** the **/proc/filesystems** entry displays the file system types known to the kernel. File systems can be loaded and unloaded dynamically as kernel modules. The contents of /proc/filesystems list only file system types that either are statically linked into the kernel or are currently loaded.
- The **/proc/locks** contains information on locks that are held on open files.
  - It is generated by the **get_locks_status()** function in */usr/src/linux/fs/locks.c*.
  - Each line represents lock information for a specific file, and documents the type of lock applied to the file.
  - The functions **fcntl()** and **flock()** are used to apply locks to files.
  - The kernel may also apply locks to files when needed.
- **Drives and Partitions;** the **/proc** file system includes information about devices connected to both IDE controllers and SCSI controllers (if the system includes them).
  - Each IDE device directory contains several entries providing access to identification and configuration information for the device. A few of the most useful are listed here:
    * **model** contains the device's model identification string.
    * **media** contains the device's media type. Possible values are disk, cdrom, tape, floppy, and UNKNOWN.
    * **capacity** contains the device's capacity, in 512-byte blocks.

```
$ cat /proc/ide/ide1/hdc/media
$ cat /proc/ide/ide1/hdc/model
$ cat /proc/scsi/scsi
$ cat /proc/sys/dev/cdrom/info
```

  - The **/proc/partitions** entry displays the partitions of recognized disk devices. For each partition, the output includes the major and minor device number, the number of 1024-byte blocks, and the device name corresponding to that partition.

- **Mounts;** the **/proc/mounts** file provides a summary of mounted file systems that you would normally expect in /etc/mtab. Each line corresponds to a single mount descriptor and lists the mounted device, the mount point, and other information.

  - The first element on the line is the mounted device.
  - The second element is the **mount point**, the place in the root file system at which the file system contents appear. For the root file system itself, the mount point is listed as /. For swap drives, the mount point is listed as **swap**.
  - The third element is the file system type. Currently, most GNU/Linux systems use the **ext2** file system for disk drives, but DOS or Windows drives may be mounted with other file system types, such as **fat** or **vfat**. Most CD-ROMs contain an **iso9660** file system.
  - The fourth element lists mount flags. These are options that were specified when the mount was added.

1.4 System Statistics

Two entries in /proc contain useful system statistics.

- The **/proc/loadavg** file contains information about the system load.

  - The first three numbers represent the number of active tasks on the system averaged over the last 1, 5, and 15 minutes.
  - The next entry shows the instantaneous current number of runnable tasks; processes that are currently scheduled to run rather than being blocked in a system call and the total number of processes on the system.
  - The final entry is the process ID of the process that most recently ran.

- The **/proc/uptime** file contains the length of time since the system was booted, as well as the amount of time since then that the system has been idle. Both are given as floating-point values, in seconds.

```
#include <stdio.h>
/* Summarize a duration of time to standard output.  TIME is the
   amount of time, in seconds, and LABEL is a short descriptive label.  */
void print_time (char* label, long time)
{
  /* Conversion constants.  */
  const long minute = 60;
  const long hour = minute * 60;
  const long day = hour * 24;
  /* Produce output.  */
  printf ("%s: %ld days, %ld:%02ld:%02ld\n", label, time / day,
  (time % day) / hour, (time % hour) / minute, time % minute);
}
int main ()
{
  FILE* fp;
  double uptime, idle_time;
  /* Read the system uptime and accumulated idle time from /proc/uptime.  */
  fp = fopen ("/proc/uptime", "r");
  fscanf (fp, "%lf %lf\n", &uptime, &idle_time);
  fclose (fp);
  /* Summarize it.  */
  print_time ("uptime   ", (long) uptime);
  print_time ("idle time", (long) idle_time);
  return 0;
}
```

Figure 7: Print the System Uptime and Idle Time.

```
$ cat /proc/uptime
```

The following program (see Fig. 7) extracts the uptime and idle time from the system and displays them in friendly units.

2 Secure Programming

- Multiple users and networking. Security is vital for some types of programs and certain types of environments.
- Many people can use the system at once, and they can connect to the system from remote locations. Unfortunately, with this power comes

risk, especially for systems connected to the Internet. Security is important, however, for a much wider range of programs and environments than people often realize.

- Crackers cover their tracks by breaking into insecure systems and using them to launch attacks against other systems.
- Crackers regularly install password sniffers on compromised machines and use the intercepted passwords to log in to and compromise other systems, repeating the process ad-infinitum.

2.1 Types of Applications

The following sections briefly cover some of the many different types of programs that can be exploited.

- *Setuid Programs;*
  - The UNIX security model relies heavily on providing access to privileged services through trusted programs that are run by ordinary users but execute with all the privileges of a more powerful user.
  - The files storing these executables have the setuid or setgid bits set, which gives the program all the privileges of the file owner (often root).
  - The untrusted user, however, has control of the arguments, data files, and environment variables used by the program. The user can control the path used by the program to search for other programs it may execute or for shared libraries it may rely on (fortunately many systems, including Linux, will override the shared library path for setuid programs).
  - The user can control when the program runs and when it is prematurely terminated.
- *Network Servers (Daemons);*
  - A cracker can easily launch a series of attacks against a server until one of them succeeds.
  - The attacker has full control of the data sent to the server and when it is sent. The attacker can cause signals to be sent over a tcp stream.

- *Network Clients;*
  - Although clients are not as easy to attack as servers (because they establish connections at their convenience and not the attacker's), they tend to be very vulnerable.
  - Many clients are very large and complicated programs, and much less attention has been given to security than for a typical server.
  - Web browsers often permit the server to execute code on the client machine (Java, JavaScript, ActiveX, and so on).
- *Mail User Agents;*
  - Mail User Agents (MUAs) can be targeted directly, particularly with buffer overflow type exploits. And attachments to messages may contain hostile programs or documents that contain macro viruses.
- *CGI Programs;*
  - CGI programs are invoked by the Web server at the request of an HTTP client to handle certain queries, form submissions, or even dynamic generation of entire Web sites.
  - They have most of the vulnerabilities that would be associated with a server. Further, CGI programs are often written in very insecure scripting languages.
- *Utilities;*
  - General utility programs that are not setuid are often thought of as not having security implications. Unfortunately, they may be used in contexts where that is not the case.
  - Most UNIX compatible systems used to run the *find* utility periodically to locate old temporary files and core files and delete them. When find executes another program with the files it has located, it passes the command, including a filename, to the shell blindly.
  - Special characters in a filename would be interpreted by the shell, resulting in the ability to execute arbitrary commands.
  - Other programs that are driven by the names of existing files and invoke other programs may have similar vulnerabilities unless precautions are taken.

  - Consider what happens when you unpack a tar archive containing a few extra files with names like /etc/passwd or /etc/rhosts. Buffer overflows are theoretically possible in the filename or other header fields.
  - Simple utilities, such as fgrep, cut, head, and tail, might be called from a CGI program. If these programs were vulnerable to buffer overflows as a result of input patterns, a system using them would be vulnerable to compromise.

- *Applications;*
  - Applications such as word processors and spreadsheets are vulnerable as well. The simple fact of the matter is that people will receive data files from untrustworthy sources.
  - Many of these have macro capabilities and the ability to have a macro automatically execute when a file is loaded; thus the popular "macro virus" was born.
  - Even worse, some systems have mail and Web browsers configured to automatically open the application whenever a data file of that type is received. Far more subtle exploits are possible using buffer overflows, for example.

2.2 Users and Groups

- The system converts your username to a particular user ID, and from then on it's only the user ID that counts.
- You can control access to a file or other resource by associating it with a particular user ID.
- Sometimes, you want to share a resource among multiple users.
- Linux doesn't allow you to associate multiple user IDs with a file, so you can't just create a list of all the people to whom you want to give access and attach them all to the file.
- You can, however, create a *group*. Each group is assigned a unique number, called a *group ID*, or *GID*. Every group contains one or more user IDs.
- A single user ID can be a member of lots of groups, but groups can't contain other groups.

- Like users, groups have names. Also like usernames, however, the group names don't really matter; the system always uses the group ID internally.
- You can associate only one group with a resource.

```
$ id
```

- One user account is very special. This user has user ID 0 and usually has the username **root**. The root user can do just about anything.
- Lots of special operations can be performed only by processes running with root privilege.
- The trouble with this design is that a lot of programs need to be run by root because a lot of programs need to perform one of these special operations. If any of these programs misbehaves, chaos can result.
- There's no effective way to contain a program when it's run by root; it can do anything. Programs run by root must be written very carefully.

2.3 Process User IDs and Process Group IDs

- The system knows only which user ID is in use, not which user is typing the commands.
- Every process has an associated user ID and group ID. When you invoke a command, it typically runs in a process whose user and group IDs are the same as your user and group IDs.
  - When we say that a user performs an operation, we really mean that a process with the corresponding user ID performs that operation.
  - When the process makes a system call, the kernel decides whether to allow the operation to proceed. It makes that determination by examining the permissions associated with the resources that the process is trying to access and by checking the user ID and group ID associated with the process trying to perform the action.
- The program in Fig. 8 shows a simple program that provides a subset of the functionality provide by the id command:

```
$ ./simpleid
```

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
  uid_t uid = geteuid ();
  gid_t gid = getegid ();
  printf ("uid=%d gid=%d\n", uid, gid);
  return 0;
}
```

Figure 8: Print User and Group IDs.

2.4 File System Permissions

- By examining how the system associates permissions with each file and then seeing how the kernel checks to see who is allowed to access which files, the concepts of user ID and group ID should become clearer.
- Each file has exactly one *owning user* and exactly one *owning group*. When you create a new file, the file is owned by the user and group of the creating process.
- You can view these *permission bits* interactively with the **ls** command by using the **-l** or **-o** options and programmatically with the **stat** system call.
- This **stat** function takes two parameters: the name of the file you want to find out about, and the address of a data structure that is filled in with information about the file.
- The program in Fig. 9 shows an example of using stat to obtain file permissions.

  ```
  $ ./stat-perm hello
  ```

- The *S_IWUSR* constant corresponds to write permission for the owning user. (*S_IRGRP, S_IXOTH*)

  ```
  chmod ("hello", S_IRUSR | S_IXUSR);
  ```

- The same permission bits apply to directories, but they have different meanings.

```
#include <stdio.h>
#include <sys/stat.h>
int main (int argc, char* argv[])
{
  const char* const filename = argv[1];
  struct stat buf;
  /* Get file information.  */
  stat (filename, &buf);
  /* If the permissions are set such that the file's
     owner can write to it, print a message.  */
  if (buf.st_mode & S_IWUSR)
    printf ("Owning user can write `%s'.\n", filename);
  return 0;
}
```

Figure 9: Determine File Owner's Write Permission.

  - If a user is allowed to read from a directory, the user is allowed to see the list of files that are present in that directory.
  - If a user is allowed to write to a directory, the user is allowed to add or remove files from the directory. Note that a user may remove files from a directory if she is allowed to write to the directory, *even if she does not have permission to modify the file she is removing.*
  - If a user is allowed to execute a directory, the user is allowed to enter that directory and access the files therein. Without execute access to a directory, a user is not allowed to access the files in that directory independent of the permissions on the files themselves.

- To summarize, let's review how the kernel decides whether to allow a process to access a particular file. It checks to see whether the accessing user is the owning user, a member of the owning group, or someone else.
- Then the kernel checks the operation that is being performed against the permission bits that apply to this user.

2.5 Security Hole: Programs Without Execute Permissions

- Here's a first example of where security gets very tricky. You might think that if you disallow execution of a program, then nobody can run it. After all, that's what it means to disallow execution.

- But a malicious user can make a copy of the program, change the permissions to make it executable, and then run the copy! If you rely on users not being able to run programs that aren't executable but then don't prevent them from copying the programs, you have a security hole means by which users can perform some action that you didn't intend.

2.5.1 Sticky Bits

- In addition to read, write, and execute permissions, there is a magic bit called the sticky bit. This bit applies only to directories.
- A directory that has the sticky bit set allows you to delete a file only if you are the owner of the file. As mentioned previously, you can ordinarily delete a file if you have write access to the directory that contains it, even if you are not the file's owner.
- When the sticky bit is set, you still must have write access to the directory, but you must also be the owner of the file that you want to delete.
- A few directories on the typical GNU/Linux system have the sticky bit set. For example, the */tmp* directory, in which any user can place temporary files, has the sticky bit set.
- Only the owning user (or root, of course) can remove a file.

```
$ ls -ld /tmp
```

- The corresponding constant to use with stat and chmod is *S_ISVTX* (Save text image after execution).
- If your program creates directories that behave like */tmp*, in that lots of people put things there but shouldn't be able to remove each other's files, then you should set the sticky bit on the directory.

```
$ chmod o+t directory
chmod (dir_path, S_IRWXU | S_IRWXG | S_IRWXO | S_ISVTX);
```

2.6 Real and Effective IDs

- Every process really has two user IDs: the *effective user ID* and the *real user ID*.

- Most of the time, the kernel checks only the effective user ID. For example, if a process tries to open a file, the kernel checks the effective user ID when deciding whether to let the process access the file.
- The **geteuid** and **getegid** functions described previously return the effective user ID and the effective group ID. Corresponding **getuid** and **getgid** functions return the real user ID and real group ID.
- There is one very important case in which the real user ID matters. If you want to change the effective user ID of an already running process, the kernel looks at the real user ID as well as the effective user ID.
- Suppose that there's a server process that might need to look at *any* file on the system, regardless of the user who created it.
- The server process could carefully examine the permissions associated with the files in question and try to decide whether *user* should be allowed to access those files.
  - But that would mean duplicating all the processing that the kernel would normally do to check file access permissions. Reimplementing that logic would be complex, errorprone, and tedious.
- A better approach is simply to temporarily change the effective user ID of the process from *root* to *user* and then try to perform the operations required.
  - If *user* is not allowed to access the data, the kernel will prevent the process from doing so and will return appropriate indications of error.
  - After all the operations taken on behalf of *user* are complete, the process can restore its original effective user ID to *root*.
- When the user enters a username and password, the login program verifies the username and password in the system password database.
- Then the login program changes both the effective user ID and the real ID to be that of the user.
- Finally, the login program calls **exec** to start the user's shell, leaving the user running a shell whose effective user ID and real user ID are that of the user.
- The function used to change the user IDs for a process is **setreuid**.

```
setreuid (geteuid(), getuid ());
```

- If a process were allowed to change its effective user ID at will, then any user could easily impersonate any other user, simply by changing the effective user ID of one of his processes.
- The kernel will let a process running with an effective user ID of 0 change its user IDs as it sees fit.
- Any other process, however, can do only one of the following things:
  - Set its effective user ID to be the same as its real user ID.
    Would be used by our accounting process when it has finished accessing files as *user* and wants to return to being root.
  - Set its real user ID to be the same as its effective user ID.
    Could be used by a login program after it has set the effective user ID to that of the user who just logged in. Setting the real user ID ensures that the user will never be able go back to being *root*.
  - Swap the two user IDs.
    Swapping the two user IDs is almost a historical artifact; modern programs rarely use this functionality.

2.6.1 Setuid Programs

```
$ whoami
user
$ su
Password: ...
$ whoami
root
```

- The **whoami** command is just like **id**, except that it shows only the effective user ID, not all the other information.
- The **su** command enables you to become the superuser if you know the root password.
- The trick is that the **su** program is a **setuid** program. That means that when it is run, the effective user ID of the process will be that of the file's owner rather than the effective user ID of the process that performed the **exec** call. (The real user ID will still be that of the executing user.)

```
#include <stdio.h>
#include <unistd.h>
int main ()
{
  printf ("uid=%d euid=%d\n", (int) getuid (), (int) geteuid ());
  return 0;
}
```

Figure 10: Setuid Demonstration Program.

- To create a **setuid** program, you use *chmod +s* at the command line, or use the *S_ISUID* flag if calling **chmod** programmatically.
- For example, consider the program in Fig. 10

```
$ ./setuid-test
$ chmod +s setuid-test
$ ls -la
$ ./setuid-test
$ ls -la
$ su
$ ./setuid-test
```

- Note that the effective user ID is set to 0 when the program is run.
- **su** is capable of changing the effective user ID through this mechanism. It runs initially with an effective user ID of 0.
- Then it prompts you for a password. If the password matches the *root* password, it sets its real user ID to be *root* as well and then starts a new shell.
- Take a look at the permissions on the **su** program:

```
$ ls -l /bin/su
```

- Notice that it's owned by *root* and that the *setuid* bit is set.
- Note that **su** doesn't actually change the user ID of the shell from which it was run. Instead, it starts a new shell process with the new user ID. The original shell is blocked until the new shell completes and **su** exits.

2.7 More Security Holes

- Although this lecture will point out a few common security holes, a great many have already been discovered, and many more are out there waiting to be found.

2.7.1 Buffer Overruns

- Almost every major Internet application daemon, including the **sendmail** daemon, the **finger** daemon, the **talk** daemon, and others, has at one point been compromised through a *buffer overrun*.
- If you are writing
  - any code that will ever be run as root,
  - a program that performs any kind of interprocess communication,
  - a program that reads files.

  you should be aware of this kind of security hole.
- The idea behind a buffer overrun attack is to trick a program into executing code that it did not intend to execute.
- The usual mechanism for achieving this feat is to overwrite some portion of the program's process stack.
  - The program's stack contains, among other things, the memory location to which the program will transfer control when the current function returns.
  - Therefore, if you can put the code that you want to have executed into memory somewhere and then change the return address to point to that piece of memory, you can cause the program to execute anything.
  - When the program returns from the function it is executing, it will jump to the new code and execute whatever is there, running with the privileges of the current process.
- If the program is running as a daemon and listening for incoming network connections, the situation is even worse. A daemon typically runs as *root*.
- A program that does not engage in network communications is much safer because only users who are already able to log in to the computer running the program are able to attack it.

- The buggy versions of **finger**, **talk**, and **sendmail** all shared a common flaw. Each used a fixed-length string buffer, which implied a constant upper limit on the size of the string but then allowed network clients to provide strings that overflowed the buffer.
- For example, they contained code similar to this:

```
 #include <stdio.h>
int main ()
{
/* Nobody in their right mind would have more than 32
characters in their username. Plus, I think UNIX allows
only 8-character usernames. So, this should be plenty
of space. */
char username[32];
/* Prompt the user for the username. */
printf ("Enter your username: ");
/* Read a line of input. */
gets (username);
/* Do other things here... */
return 0;
}
```

- The combination of the 32-character buffer with the **gets** function permits a buffer overrun.
- The **gets** function reads user input up until the next newline character and stores the entire result in the *username* buffer.
  - The attacker might deliberately type in a very long *username*.
  - Local variables such as *username* are stored on the *stack*, so by exceeding the array bounds, it's possible to put arbitrary bytes onto the stack beyond the area reserved for the username variable.
  - The username will overrun the buffer and overwrite parts of the surrounding *stack*, allowing the kind of attack described previously.
- Fortunately, it's relatively easy to prevent buffer overruns. When reading strings, you should always use a function, such as **getline**, that either dynamically allocates a sufficiently large buffer or stops reading input if the buffer is full. For example, you could use this:

```
#include <stdlib.h>
#include <stdio.h>
main()
{
char command[80]="date";
char name[20];
printf("Please enter your name: ");
gets(name);
printf("Hello, %s, the current date and time is: ",name);
fflush(stdout);
system(command);
}
```

Figure 11: A Simple Buffer Overflow Program and Its Execution.

```
char* username = getline (NULL, 0, stdin);
```

This call automatically uses **malloc** to allocate a buffer big enough to hold the line and returns it to you. You have to remember to call **free** to deallocate the buffer, of course, to avoid leaking memory.

- Of course, buffer overruns can occur with any statically sized array, not just with strings.
- If you want to write secure code, you should never write into a data structure, on the stack or elsewhere, without verifying that you're not going to write beyond its region of memory.
- In the simplest case, a buffer overflow allows the user supplying input to one untrusted variable to overwrite another variable, which is assumed to be safe from untrusted input. Imagine the trivial program in Fig. 11 running with stdin/stdout connected to an outside source through some means. then execute like

```
./bufferoverflow
Please enter your name: 12345678whoami
Hello, 12345678whoami, the current date and time is: ozdogan
```

  - It turns out that "command" is stored in memory right above "name".
  - Local variables in most implementations of C and many other languages are stored next to each other on the stack. Of course,

Table 1: Vulnerable Standard Library Functions and Alternatives

Bad Syntax	Better Syntax	Notes
gets()	fgets()	Different handling of newlines may leave unread characters in stream
sprintf()	snprintf()	Not available on many other OSes
vsprintf()	vsnprintf()	Not available on many other OSes
strcpy()	strncpy()	Omits trailing null if there is an overflow
strcat()	strncat()	Omits trailing null if there is an overflow
stpcpy()	stpncpy()	Copies exactly the specified size of characters into the target

crackers trying to repeat this feat with other programs often found the variables they were overwriting were too mundane to be of much use.

- They could only usefully overwrite those variables declared in the same function before the variable in question.
- True, the variables for the function that called this function were just a little further along on the stack, as well as the function which called it, and the function before it; but before you overwrote those variables you would overwrite the return address in between and the program would never make it back to those functions.
- You see, the return address is a very important value. By overwriting its value, you can transfer control of the program to any existing section of code.
- Of course, you still need an existing section of code that will suit your evil purposes and you need to know exactly where it is located in memory on the target system.
- Dissatisfied with being limited to the existing code? Well, why not supply your own binary executable code in the string itself?

- If you must execute a program with any untrusted user input as arguments, use one of the **exec()** family of calls instead of **system()**.
- For vulnerable standard library functions and alternatives see Table 1

2.7.2 Race Conditions in /tmp

- Another very common problem involves the creation of files with predictable names, typically in the */tmp* directory.
  - Suppose that your program **prog**, running as *root*, always creates a temporary file called */tmp/prog* and writes some vital information there.
  - A malicious user can create a symbolic link from */tmp/prog* to any other file on the system.
  - When your program goes to create the file, the **open** system call will succeed.
  - However, the data that you write will not go to */tmp/prog*; instead, it will be written to some arbitrary file of the attacker's choosing.
- This kind of attack is said to exploit a *race condition*. There is implicitly a race between you and the attacker. Whoever manages to create the file first *wins*.
- One attempt at avoiding this attack is to use a randomized name for the file. For example, you could read from */dev/random* to get some bits to use in the name of the file.
- This certainly makes it harder for a malicious user to guess the filename, but it doesn't make it impossible. The attacker might just create a large number of symbolic links, using many potential names. Even if she has to try 10,000 times before wining the race condition.
- Another approach is to use the *O_EXCL* flag when calling **open**. This flag causes open to fail if the file already exists.
- Unfortunately, if you're using the Network File System (NFS), or if anyone who's using your program might ever be using NFS, that's not a sufficiently robust approach because *O_EXCL* is not reliable when NFS is in use.
- One approach that works is to call **lstat** on the newly created file.
- The **lstat** function is like **stat**, except that if the file referred to is a symbolic link, **lstat** tells you about the link, not the file to which it refers.

- If **lstat** tells you that your new file is an ordinary file, not a symbolic link, and that it is owned by you, then you should be okay.
- This attack doesn't create any direct harm, but it does make it impossible for our program to get its work done. Such an attack is called a *denial-of-service (DoS)* attack.
- The program in Fig. 12 presents a function that tries to securely open a file in */tmp*.
- This function calls **open** to create the file and then calls **lstat** a few lines later to make sure that the file is not a symbolic link.

2.7.3 Using system or popen

- The third common security hole that every programmer should bear in mind involves using the shell to execute other programs.
- Let's consider a dictionary server. This program is designed to accept connections via internet.
- Each client sends a word, and the server tells it whether that is a valid English word.

```
$ grep -x word /usr/dict/words
```

- The program in Fig. 13 shows how you might try to code the part of the server that invokes grep:
- Note that by calculating the number of characters we need and then allocating the buffer dynamically, we're sure to be safe from buffer overruns.
- Unfortunately, the use of the **system** function is unsafe. This function invokes the standard system shell to run the command and then returns the exit value.
  - But what happens if a malicious hacker sends a "word" that is actually the following line or a similar string?

    ```
    foo /dev/null; rm -rf /
    grep -x foo /dev/null; rm -rf / /usr/dict/words
    ```

- The same problem can arise with **popen**, which creates a pipe between the parent and child process but still uses the shell to run the command.

```
#include <fcntl.h> #include <stdio.h> #include <stdlib.h>
#include <sys/stat.h> #include <unistd.h>
/* Returns the file descriptor for a newly created temporary file. The temporary
file will be readable and writable by the effective user ID of the current
process but will not be readable or writable  by anybody else.
   Returns -1 if the temporary file could not be created.  */
int secure_temp_file ()
{  /* This file descriptor points to /dev/random and allows us to get a
     good source of random bits.  */
  static int random_fd = -1;
  /* A random integer.  */
  unsigned int random;
  /* A buffer, used to convert from a numeric to a string representation of
  random.  This buffer has fixed size, meaning that we potentially have a
  buffer overrun bug if the integers on this machine have a *lot* of bits.  */
  char filename[128];
  /* The file descriptor for the new temporary file.  */
  int fd;
  /* Information about the newly created file.  */
  struct stat stat_buf;
  /* If we haven't opened /dev/random, do so now.  (This is not threadsafe.)  */
  if (random_fd == -1) {
/* Open /dev/random.  Note that we're assuming that /dev/random really is a source
of random bits, not a file full of zeros placed there by an attacker.  */
    random_fd = open ("/dev/random", O_RDONLY);
    /* If we couldn't open /dev/random, give up.  */
    if (random_fd == -1)
      return -1;  }
  /* Read an integer from /dev/random.  */
  if (read (random_fd, &random, sizeof (random)) !=
      sizeof (random))
    return -1;
  /* Create a filename out of the random number.  */
  sprintf (filename, "/tmp/%u", random);
  /* Try to open the file.  */
  fd = open (filename,
     /* Use O_EXECL, even though it doesn't work under NFS.  */
     O_RDWR | O_CREAT | O_EXCL,
     /* Make sure nobody else can read or write the file.  */
     S_IRUSR | S_IWUSR);
  if (fd == -1)
    return -1;
  /* Call lstat on the file, to make sure that it is not a symbolic link.  */
  if (lstat (filename, &stat_buf) == -1)
    return -1;
  /* If the file is not a regular file, someone has tried to trick us.  */
  if (!S_ISREG (stat_buf.st_mode))
    return -1;
  /* If we don't own the file, someone else might remove it, read it,
     or change it while we're looking at it.  */
  if (stat_buf.st_uid != geteuid () || stat_buf.st_gid != getegid ())
    return -1;
/* If there are any more permission bits set on the file, something's fishy.*/
  if ((stat_buf.st_mode & ~(S_IRUSR | S_IWUSR)) != 0)
    return -1;
  return fd;}
```

Figure 12: Create a Temporary File.

```
#include <stdio.h>
#include <stdlib.h>
/* Returns a non-zero value if and only if the WORD appears in
   /usr/dict/words.  */
int grep_for_word (const char* word)
{
  size_t length;
  char* buffer;
  int exit_code;
  /* Build up the string ‘grep -x WORD /usr/dict/words’.  Allocate the
     string dynamically to avoid buffer overruns.  */
  length =
    strlen ("grep -x ") + strlen (word) + strlen (" /usr/dict/words") + 1;
  buffer = (char*) malloc (length);
  sprintf (buffer, "grep -x %s /usr/dict/words", word);
  /* Run the command.  */
  exit_code = system (buffer);
  /* Free the buffer.  */
  free (buffer);
  /* If grep returned zero, then the word was present in the
     dictionary.  */
  return exit_code == 0;
}
```

Figure 13: Search for a Word in the Dictionary.

- There are two ways to avoid these problems.
  - One is to use the **exec** family of functions instead of **system** or **popen**. That solution avoids the problem because characters that the shell treats specially (such as the semicolon in the previous command) are not treated specially when they appear in the argument list to an exec call.
  - The other alternative is to validate the string. In the dictionary server example, you would make sure that the word provided contains only alphabetic characters, using the **isalpha** function. If it doesn't contain any other characters, there's no way to trick the shell into executing a second command.