Following is all the information that you need to understand the workings of
the UNIX operating system (Berkley 4.2).

Patched together by The War

On the security side of UNIX:

On the Security of UNIX Dennis M. Ritchie Recently there has been much interest
in the security aspects of operating systems and software. At issue is the
ability to prevent undesired disclosure of information, destruction of
information, and harm to the functioning of the system. This paper discusses
the degree of security which can be provided under the system and offers a
number of hints on how to improve security. The first fact to face is that was
not developed with security, in any realistic sense, in mind; this fact alone
guarantees a vast number of holes. (Actually the same statement can be made
with respect to most systems.) The area of security in which is theoretically
weakest is in protecting against crashing or at least crippling the operation
of the system.
   The problem here is not mainly in uncritical acceptance of bad parameters
to system calls there may be bugs in this area, but none are known- but rather
in lack of checks for excessive consumption of resources. Most notably, there
is no limit on the amount of disk storage used, either in total space allocated
or in the number of files or directories. Here is a particularly ghastly shell
sequence guaranteed to stop the system:

```
    while :; do
        mkdir x
        cd x
     done
```

Ether a panic will occur because all the i-nodes on the device are used up,
or all the disk blocks will be consumed, thus preventing anyone from
writing files on the device. In this version of the system, users are
prevented from creating more than a set number of processes simultaneously, so
unless users are in collusion it is unlikely that any one can stop the
system altogether. However, creation of 20 or so CPU or disk-bound jobs
leaves few resources available for others. Also, if many large jobs are
run simultaneously, swap space may run out, causing a panic. It should be
evident that excessive consumption of disk space, files, swap space, and
processes can easily occur accidentally in malfunctioning programs
as well as at command level. In fact is essentially defenseless against
□this kind of abuse, nor is there any easy fix. The best that can be said is
that it is generally fairly easy to detect what has happened when disaster
strikes, to identify the user responsible, and take appropriate action.
In practice, we have found that difficulties in this area are rather rare,
but we have not been faced with malicious users, and enjoy a fairly generous
supply of resources which have served to cushion us against accidental
overconsumption. The picture is considerably brighter in the area of protection
of information from unauthorized perusal and destruction. Here the degree of
security seems (almost) adequate theoretically, and the problems lie more in
the necessity for care in the actual use of the system. Each file has
associated with it eleven bits of protection information together
with a user identification number and a usergroup identification number (UID
and GID). Nine of the protection bits are used to specify independently
permission to read, to write, and to execute the file to the user himself,
to members of the user's group, and to all other users. Each process
generated by or for a user has associated with it an effective UID and
a real UID, and an effective and real GID. When an attempt is made to access
the file for reading, writing, or execution, the user process's effective
UID is compared against the file's UID; if a match is obtained, access is
granted provided the read, write, or execute bit respectively for the user
himself is present. If the UID for the file and for the process fail to
match, but the GID's do match, the group bits are used; if the GID's do

not match, the bits for other users are tested. The last two bits of each
file's protection information, called the set-UID and set-GID bits, are used
only when the file is executed as a program. If, in this case, the set-UID
bit is on for the file, the effective UID for the process is changed to the
UID associated with the file; the change persists until the process
☐terminates or until the UID changed again by another execution of a set-UID
file. Similarly the effective group ID of a process is changed to the GID
associated with a file when that file is executed and has the set-GID bit
set. The real UID and GID of a process do not change when any file is
executed, but only as the result of a privileged system call. The basic
notion of the set-UID and set-GID bits is that one may write a program which
is executable by others and which maintains files accessible to others
only by that program. The classical example is the game-playing program
which maintains records of the scores of its players. The program itself has
to read and write the score file, but no one but the game's sponsor can be
allowed unrestricted access to the file lest they manipulate the game to their
own advantage. The solution is to turn on the set-UID bit of the game program.
When, and only when, it is invoked by players of the game, it may update the
score file but ordinary programs executed by others cannot access the
score. There are a number of special cases involved in determining access
permissions. Since executing a directory as a program is a meaningless
operation, the execute-permission bit, for directories, is taken instead to
mean permission to earch he directory for a given file during the scanning of
a path name; thus if a directory has execute permission but no read
permission for a given user, he may access files with known names in the
directory, but may not read (list) the entire contents of the directory. Write
permission on a directory is interpreted to mean that the user may
create and delete files in that directory; it is impossible for any
user to write directly into any directory. Another, and from the point
of view of security, much more serious special case is that there is a ``super
user'' who is able to read any file and write any nondirectory. The
super-user is also able to change the protection mode and the owner UID and
☐GID of any file and to invoke privileged system calls. It must be
recognized that the mere notion of a super-user is a theoretical, and
usually practical, blemish on any protection scheme. The first necessity
for a secure system is of course arranging that all files and
directories have the proper protection modes. Traditionally, software has been
exceedingly permissive in this regard; essentially all commands create files
readable and writable by everyone. In the current version, this policy may be
easily adjusted to suit the needs of the installation or the individual
user. Associated with each process and its descendants is a mask, which is in
effect with the mode of every file and directory created by that process. In
this way, users can arrange that, by default, all their files are no more
accessible than they wish. The standard mask, set by allows all
permissions to the user himself and to his group, but disallows writing by
others. To maintain both data privacy and data integrity, it is necessary,
and largely sufficient, to make one's files inaccessible to others. The lack
of sufficiency could follow from the existence of set-UID programs created
by the user and the possibility of total breach of system security in one
of the ways discussed below (or one of the ways not discussed below). For
greater protection, an encryption scheme is available. Since the editor
is able to create encrypted documents, and the command can be used to pipe
such documents into the other text-processing programs, the length of time
during which cleartext versions need be available is strictly limited. The
encryption scheme used is not one of the strongest known, but it is judged
adequate, in the sense that cryptanalysis is likely to require
considerably more effort than more direct methods of reading the encrypted
files. For example, a user who stores data that he regards as truly secret
should be aware that he is implicitly trusting the system administrator not
☐to install a version of the crypt command that stores every typed
password in a file. Needless to say, the system administrators must be at
least as careful as their most demanding user to place the correct
protection mode on the files under their control. In particular, it is
necessary that special files be protected from writing, and probably

reading, by ordinary users when they store sensitive files belonging to
other users. It is easy to write programs that examine and change files
by accessing the device on which the files live. On the issue of password
security, is probably better than most systems. Passwords are stored in an
encrypted form which, in the absence of serious attention from specialists in
the field, appears reasonably secure, provided its limitations are
understood. In the current version, it is based on a slightly defective
version of the Federal DES; it is purposely defective so that easily-
available hardware is useless for attempts at exhaustive key-search.
Since both the encryption algorithm and the encrypted passwords are available,
exhaustive enumeration of potential passwords is still feasible up to a
point. We have observed that users choose passwords that are easy to
guess: they are short, or from a limited alphabet, or in a dictionary.
Passwords should be at least six characters long and randomly chosen
from an alphabet which includes digits and special characters. Of course
there also exist feasible non-cryptanalytic ways of finding out
passwords. For example: write a program which types out ``login:'' on
the typewriter and copies whatever is typed to a file of your own. Then
invoke the command and go away until the victim arrives.   The set-UID (
set-GID) notion must be used carefully if any security is to be maintained.
The first thing to keep in mind is that a writable set-UID file can have
another program copied onto it. For example, if the super-user command is
□writable, anyone can copy the shell onto it and get a password-free version
of A more subtle problem can come from set-UID programs which are not
sufficiently careful of what is fed into them. To take an obsolete
example, the previous version of the command was set-UID and owned by the
super-user. This version sent mail to the recipient's own directory. The
notion was that one should be able to send mail to anyone even if they want
to protect their directories from writing. The trouble was that was
rather dumb: anyone could mail someone else's private file to himself. Much
more serious is the following scenario: make a file with a line like one in
the password file which allows one to log in as the super-user. Then make a
link named ``.mail'' to the password file in some writable directory on the
same device as the password file (say/tmp). Finally mail the bogus login
line to /tmp/.mail; You can then login as the superuser, clean up the
incriminating evidence, and have your will. The fact that users can mount
their own disks and tapes as file systems can be another way of gaining
superuser status. Once a disk pack is mounted, the system believes what is on
it. Thus one can take a blank disk pack, put on it anything desired, and
mount it. There are obvious and unfortunate consequences.   For
example: a mounted disk with garbage onit will crash the system; one of the
files on the mounted disk can easily be a password-free version of other
files can be unprotected entries for special files. The only easy fix for
this problem is to forbid the use of to unprivileged users. A partial
solution, not so restrictive, would be to have the command examine the
special file for bad data, set-UID programs owned by others, and
accessible special files, and balk at unprivileged invokers.

-
Info about the /etc/passwd file:
□---

NME
     passwd - password file

DSCRIPTION
     Passwd contains for each user the
following information:

    name (login name, contains no
upper case)
     encrypted password
     numerical user ID
     numerical group ID

user's real name, office,
extension, home phone.
     initial working directory
     program to use as Shell

   The name may contain `&', meaning insert the login name.
    This information is set by the chfn(1) command and used by
    the finger(1) command.

   This is an ASCII file. Each field within each user's entry
    is separated from the next by a colon. Each user is
    separated from the next by a new line. If the password
    field is null, no password is demanded; if the Shell field
    is null, then /bin/sh is used.

   This file resides in directory / etc. Because of the
    encrypted passwords, it can and does have general read
    permission and can be used, for example, to map numerical user
    ID's to names.

   Appropriate precautions must be taken to lock the file
    against changes if it is to be edited with a text editor;
    vipw(8) does the necessary locking.

FLES
     /etc/passwd

SE ALSO
☐   getpwent(3), login(1), crypt(3),
passwd(1), group(5),
     chfn(1), finger(1), vipw(8),
adduser(8)

BGS
     A binary indexed file format should be available for fast access.

   User information (name, office, etc.) should be stored elsewhere.

   Now if you have had the patience to read all of this and you have digested
it you know everything that you need to know about the Unix system to hold up
your end of an intelligent conversation.

Have fun!