*** A List Of Some OF The Most Useful UNIX **
*** Hacking Commands, and Some Hints On Their Usage ***

It is fun and often usefull to create a file that is owned by someone else. On most systems with slack security ie 99% of all UNIX systems, this is quite easily done. The chown command will change any of your files to make someone else the owner. Format is as follows:

chown ownername filelist

Where ownername is the new owner, and filelist is the list of files to change. You must own the file which your are goin to change, unless you are a superuser....then u can change ANYTHING!
chgrp is a similar command which will change the group ownership on a file. If you are going to do both a chown and a chgrp on a file, then make sure you do the chgrp first! Once the file is owned by someone else, you cant change nything about it!

Sometimes just seeing who is on the system is a challenge in itself. The best way is to write your own version of who in C, but if you can't do that then this may be of some help to you:

who followed by on or more of the following flags:

-b Displays time sys as last booted.
-H Precedes output with header.
-l Lists lines waiting for users to logon.
-q displays number of users logged on.
-t displays time sys clock was last changed.
-T displays the state field (a + indicates it is possible to send to terminal, a - means u cannot)
-u Give a complete listing of those logged on.

who -HTu is about the best choice for the average user

##by the way, the list of users logged on is kept in the file /etc/utmp. If you want to write your own personalised version of who in C, you now know where to look!###

When a users state field (see -T flag option for who command) says that a user has their message function on, this actually means that it is possible to get stuff onto their screen.

Basically, every terminal on the system has a file corresponding to it. These files can be found in the /dev directory. You can to anything to these files, so long as you have access -eg you can read them, and write to them, but you will notice that they never change in size. They are called character specific files, and are really the link between the system and the terminals. Whatever you put in these files will go staright to the terminal it corresponds to.

Unfortunately, on most systems, when the user logs in, the "mesg n" command is issued which turns off write access to that terminal, BUT- if you can start cating to that terminal before system issues the mesg n command, then you will continue to be able to get stuff up on that terminal! This has many varied uses.

Check out the terminal, or terminal software being used. Often you will be able to remotely program another users terminal, simply by 'cating' a string to a users screen. You might be able to set up a buffer, capturing all that is typed, or you may be able to send the terminal into a frenzy- (sometimes a user will walk away without realizing that they are sill effectively logged on, leaving you with access to their account!). Some terminal types also have this great command called transmit screen. It transmits everything on the screen, just as if the user had typed it !

So just say I wanted to log off a user, then I would send a clear screen command (usually ctrl l), followed by "exit" followed by a carriage return, followed by the transmit screen code. Using ths technique you can wipe peoples directories or anything. My favourite is to set open access on all their files and directories so I can peruse them for deletion etc at my own leisure).

If you ever briefly get access to another persons account eg. they leave the room to go to toilet or whatever, then simply type the following:

```
chmod 777 $HOME
chmod 777 $MAIL
```

Then clear the screen so they dont see what you just typed.

Now you can go look at their directory, and their mail, and you can even put mail in their mail file. (just use the same

format as any mail that is already there!). Next time they log in the system will automatically inform them they have new mail!

Another way to send fake mail to people is to use the mail server. This method produces mail that is slightly different to normal, so anyone who uses UNIX a bit may be suspiscious when they receive it, but it will fool the average user!

type telnet

the following prompt will appear:

telnet>

now type :

open localhost 25

some crap will come up about the mail server..now type:

mail from: xxxxxx Put any name you want.

some more bullshit will come up. Now type:

rcpt to: xxxxxx Put the name of the person to receive mail here.

now type:

data

now you can type the letter...end it with a "."
type quit to exit once you are done.

Heres one for any experimenters out there...
It is possible to create files which simply cannot be deleted from the standard shell. To do this you will have to physically CREATE THE FILE USING A C PROGRAM or SCRIPT FILE, and you will have to use a sequence of control characters which cannot be typed from the shell. Try things like Ctrl-h (this is the code for the delete key). Just a file with the name Ctrl-h would not be deleteable from the shell, unless you used wildcards. So, make it a nice long series of characters, so that to delete the file, the user has no choice but to individually copy all his files elsewhere, then delete everything in his directory, and then copy all his files back.....this is one of my favourites..gets em every time!

The following script file is an example which will create a file with the name Ctrl-h. You MUST tyoe this file in using the vi editor or similar.
*****If you are not very good with vi, type "man vi" and print the help file...it even contains stuff that I find useful now and then.*****

type the following in vi...

echo'' > 'a^h'

***NOTE...to get the ^h (this really means ctrl-h) from vi type:

Ctrl v
Ctrl h

The Ctrl v instrcts vi to take the next character as a ascii character, and not to interpret it.
change the access on the file you just created and now execute it. It will create a file which looks like it is called a, but try to delete it !..use wildcards if you really want to delete it.

*> Title: Tutorial on hacking through a UNIX system

**

In the following file, all references made to the name Unix, may also be

substituted to the Xenix operating system.

Brief history: Back in the early sixties, during the development of third generation computers at MIT, a group of programmers studying the potential of computers, discovered their ability of performing two or more tasks simultaneously. Bell Labs, taking notice of this discovery, provided funds for their developmental scientists to investigate into this new frontier. After about 2 years of developmental research, they produced an operating system they called "Unix".
Sixties to Current: During this time Bell Systems installed the Unix system to provide their computer operators with the ability to multitask so that they could become more productive, and efficient. One of the systems they put on the Unix system was called "Elmos". Through Elmos many tasks (i.e. billing,and installation records) could be done by many people using the same mainframe.

Note: Cosmos is accessed through the Elmos system.

Current: Today, with the development of micro computers, such multitasking can be achieved by a scaled down version of Unix (but just as powerful). Microsoft,seeing this development, opted to develop their own Unix like system for the IBM line of PC/XT's. Their result they called Xenix (pronounced zee-nicks). Both Unix and Xenix can be easily installed on IBM PC's and offer the same function (just 2 different vendors).

Note: Due to the many different versions of Unix (Berkley Unix, Bell System III, and System V the most popular) many commands following may/may not work. I have written them in System V routines. Unix/Xenix operating systems will be considered identical systems below.

How to tell if/if not you are on a Unix system: Unix systems are quite common systems across the country. Their security appears as such:

Login;   (or login;)
password:

When hacking on a Unix system it is best to use lowercase because the Unix system commands are all done in lower- case. Login; is a 1-8 character field. It is
usually the name (i.e. joe or fred) of the user, or initials (i.e. j.jones or f.wilson). Hints for login names can be found trashing the location of the dial-up (use your CN/A to find where the computer is). Password: is a 1-8 character password assigned by the sysop or chosen by the user.

Common default logins

login;	Password:
root	root,system,etc..
sys	sys,system
daemon	daemon
uucp	uucp
tty	tty
test	test
unix	unix
bin	bin
adm	adm
who	who
learn	learn
uuhost	uuhost
nuucp	nuucp

If you guess a login name and you are not asked for a password, and have accessed to the system, then you have what is known as a non-gifted account. If you guess a correct login and pass- word, then you have a user account. And, if you get the root p/w you have a "super-user" account.
All Unix systems have the following installed to their system:
root, sys, bin, daemon, uucp, adm Once you are in the system, you will get a prompt. Common prompts are:

$
%
#

But can be just about anything the sysop or user wants it to be.

Things to do when you are in: Some of the commands that you may want to try follow below:

who is on (shows who is currently logged on the system.)
write name (name is the person you wish to chat with)
To exit chat mode try ctrl-D.
EOT=End of Transfer.
ls -a   (list all files in current   directory.)
du -a   (checks amount of memory your files use;disk usage)
cd\name   (name is the name of the sub-directory you choose)
cd\   (brings your home directory to current use)
cat name   (name is a filename either a program or documentation your username has written)
 Most Unix programs are written in the C language or Pascal
 since Unix is a programmers' environment. One of the first things done on the

system is print up or capture (in a buffer) the file containing all user names and accounts. This can be done by doing the following command:

```
cat /etc/passwd
```

If you are successful you will see a list of all accounts on the system. It should look like this:
root:hvnsdcf:0:0:root dir:/: joe:majdnfd:1:1:Joe Cool:/bin:/bin/joe hal::1:2:Hal Smith:/bin:/bin/hal

The "root" line tells the following info :
login name=root
hvnsdcf = encrypted password
0 = user group number
0 = user number
root dir = name of user
/ = root directory

In the Joe login, the last part "/bin/joe " tells us which directory is his home directory (joe) is. In the "hal" example the login name is followed by 2 colons, that means that there is no password needed to get in using his name.

Conclusion: I hope that this file will help other novice Unix hackers obtain access to the Unix/Xenix systems that they may find.

On the Security of UNIX

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 UNIX 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
```

Either 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 diskspace, files, swap space and processes can easily occur accidentally in malfunctioning programs as well as at command level.In fact UNIX 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 UNIX file has associated with it eleven bits of protection information together with a user identification number and a user-group 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 executing 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 executableby 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 search the 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 non-directory.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, UNIX 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 ofthe installation or the individual user.

Associated with each process and its descendants is a mask, which is in effect anded 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 login,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 crypt command can be used to pipe such documents into the other text-processing programs,the length of time during which clear text 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 cryptanalysisis 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 otherusers.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,UNIX 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 slightl y 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 Shell Unix.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 mail command was set-UID and owned by the super-user.This version sent mail to the r ecipient's own directory.The notion was that one should be able to send mail to

anyone even if they want to protecttheir directories from writing. The trouble was that mailwas rather dumb:anyone could mail someone else's priva te file to himself.Much more seriousis the following scenario: make a file with a line like one in the password filewhich 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 super-user 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 on it will crash the system;one of the files on the mounted disk can easily be a password-free version of Shell Unix; other files can be unprotected entries for special files. The only easy fix for this problem is to forbid the use of mount to unpriv- ileged users.A partial solution, not so restrictive,would be to have the mount command examine the special file for bad data,set-UID programs owned by others ,and accessible special files,and balk at unprivileged invokers.

Scott Walters London, CANADA
'Beware the fury of a patient man.'