Computer Security, Lab 2.

UNIX/Linux Security
Login, File Access Schemes and Dictionary Attack.

February 4, 2011

Read this earlier than one day before the lab!

There are preparatory assignments for this lab. For most students, these assignments take more than a couple of minutes. If you are not familiar with the C programming language, you will have to learn some basics skills before you do the preparatory assignments. Read through this lab guide, and then prepare your assignments. Be generous in commenting your programs. During the lab, answer all **Problems** on a separate sheet of paper, so your work can be approved.

In order to prepare your assignments, you need to download some programs.
These programs can be found on the course webpage.
Check this page and download the programs before you start preparing. The available programs are also listed in **Appendix B**. You are going to use a password cracking program during the lab. One preparatory assignment is to read the manual for that program so that you easily can use it when you are doing the lab. However, as you might guess, **it is strictly forbidden to run this, or any other password cracking program on the LTH network computers**.

The computers in the laboratory are **not** equipped with floppy drives. They **do** have CD drives and usb stick connections so it is possible to access your solutions from the preparatory assignments in that manner.

IMPORTANT! The lab computers are completely shielded from the internet. As a consequence, any information you believe will be needed during the lab, has to be written down/printed out beforehand. Study the questions in this lab manual, consider what you will need to be able to solve them, and make sure you bring that information with you.

Goals.

In this lab you will learn how the *login* program in UNIX/Linux works. You will write a program that simulates the login procedure, using a private database(file) of users and passwords. You will also look at the file access scheme used in UNIX/Linux. In the last part of the lab you will try to attack a user account on a remote machine, by performing a dictionary attack on a password file.

Introduction.

The computers in this lab are running Ubuntu 9.10, kernel version 2.6.x. Kernel is UNIXish for core operating system. The kernel is responsible for e.g. process privileges, process scheduling, memory and file access. Apart from a few low level routines written in assembler, the Linux kernel

is written in C. Since you are going to use library functions and interact with the kernel, *your programs must be written in C*. If you do not feel totally comfortable with compiling C programs, linking libraries and using the `man` command, you can read in **Appendix A** to get started. In the appendix there are also a brief description of *pointers* and explanation of *call by reference/value* in C.

The programs and command described here are those used in Ubuntu and Debian GNU/Linux. Different flavors of UNIX have slightly different commands. Most commands will agree with the ones used in e.g. Solaris. You should be able to test most of your preparatory assignments on SUN computers as well. If you use the supplied program `pwdblib.c` on a SUN computer, you **must** include the command line option `"-DSUN"` to the gcc compiler to make it work. All files that you should download and study will be available on your computer in the directory `/usr/local/lab2/` when you do your lab.

Preparatory assignment

- Read about UNIX security in Chapter 6 in *Computer Security*, by Dieter Gollmann.
- Read the manual pages for `login(1)` and `passwd(5)`. Make sure you understand the entry format of `/etc/passwd`.

When you write your program for this lab, try to split the overall problem into smaller functions. In that way you can have a small `main()` function which is common for Part 1, and as you proceed with Part 1, the `main()` function just calls different subfunctions.

Part 1: Login.

The first problem we are going to consider is the login routine used in Linux. When you do a console login, the first program you encounter is `/sbin/getty`[1]. This program is responsible for writing the `"login:"` prompt. After the user enters a *username*, `getty` starts the `/bin/login` program and hands over the entered *username*. Then, `/bin/login` writes `"Password:"` and waits for the user to enter a *password*.

Problem 1. *Why is the text not echoed on the screen when the user enters the password?*

The entered *password* is then hashed using the `crypt(3)` function and the result is compared with the user's entry in `/etc/passwd`. If the password match, the `login` program starts the user's *shell* specified in `/etc/passwd`. The shell is started with the uid (user identity number) of the user that just logged in. For a X based (graphical) login, the daemon `xdm` (or `gdm`) provides similar services as `getty` and `login`.

Problem 2. *Does it exist some hash value that corresponds to two (or more) different passwords? What is the maximum length of a password?*

[1] In Solaris this program is called ttymon.

Preparatory assignment

- Study the downloaded program `userinfo.c`. Copy it to a new file called "mylogin.c". Rewrite it so that it simulates the system login procedure. The program should check the password entered by the user with the corresponding entry in the `/etc/passwd` file. Useful library functions might include `getpwnam(3)`, `getpass(3)`, `crypt(3)` and `strcmp(3)`. If the passwords match, the program should write something like *"User authenticated successfully"* and terminate. If the password is wrong it should respond *"Unknown user or incorrect password."* and start over with the `"login:"` prompt.

Problem 3. *Compile and run your program* `mylogin.c` *with the functionality we have described so far. Remember to link the* `crypt` *library to your program (the* `-lcrypt` *option) if you use the* `crypt(3)` *function. Does it work as you expect?*

In the first part you have been accessing the real `/etc/passwd` file. In the following, you are going to extend the functionality of this file. Thus, you need to change your program so that you access a local file instead. To make things easier, the program `/usr/local/lab2/pwdblib.c` implements the functions to access and update the local file.

- Copy `/usr/local/lab2/pwdblib.c` and `/usr/local/lab2/pwdblib.h` to your home directory.
- Make sure your current directory is your home directory. Run the program `/usr/local/lab2/pwdb_extract`. The program will extract information from `/etc/passwd`, add some extra fields in each entry, and write a local file in your current directory called `pwfile`.

Check the contents of the local `pwfile` by running `"cat pwfile"`.

Preparatory assignment

- Study the files `pwdblib.c` and `pwdblib.h`. Add functions to your program such that it can access the local password file instead of the system password file. The fields of each entry has changed, read in the header file and adapt your new functions accordingly. There are two demo programs on the homepage, which uses the functions from `pwdblib.c`. You can study them if you are uncertain on how to use `pwdblib.c`. Implement the following features:

  1) The field `pw_failed` should be used to count the number of *unsuccessful* logins. Each time an unsuccessful login is encountered this counter should increase by one. When the right password is entered, `pw_failed` should be reset to zero.

  2) The field `pw_age` should be used to count the number of *successful* logins. When the age is greater than some value, e.g. 10, the user should be reminded to change his/hers password. This field would be zeroed by a program corresponding to `passwd(1)`, which you don't have to write.

  3) Use the `pw_failed` counter to lock out a user account which has entered the wrong password more than say 5 times in a row. The account locking can be made in several ways, implement it as you like but remember that it should be easy for the administrator to enable the account again. So erasing the entry in `pwfile` is not a great idea...

Compile, debug, compile again and run your program. Check the `pwfile` from another terminal window while you are testing your program. Does it work as you expected?

Problem 4. *Why is it important to have a* `pw_failed` *counter?*
Considering an on-line attack, propose some other countermeasure that could easily be implemented.

Problem 5. *Can you think of any problems with "locking" a user account automatically as done in this exercise?*

Problem 6. *Try to press* Control-C *at various times in your program. Does the program terminate every time? When does it terminate and when does it not? Why?*

Secure your program against *Control-C*. Check the man pages for `signal(2)` and `signal(7)`. You can read more about signals in Chapter 21 in [2]. Is there any other signal you should protect your program against?

Problem 7. *What can happen if two instances of your program access the passwd file simultaneously? Is there a way of preventing simultaneous access?*

Starting a User Shell.

Normally your login program would run with setuid root, so that it could start a shell for any user. We will fix that in a while. First you need to implement the function that starts the preferred shell.

Preparatory assignment

- Study the program `openshell_demo.c`. Make sure you understand how the program works. Remember that when you call `fork(2)`, both the parent and the child process continues execution in the program, but `fork` returns different values to the parent and the child, so that you can separate the line of execution.
- Read about the functions `setuid(2)`, `seteuid(2)`, `setgid(2)` and `setegid(2)`.

There is a good description of both `fork` and `setuid` in *GNU C Library Reference Manual* [2]. You can download the complete manual or the relevant chapters from the lab homepage or find it in /usr/local/lab2.

Problem 8. *What is the difference between "real user" and "effective user" of a program? Describe the circumstances under which they are different/equal?*

Implement the following features in your program:

- After a successful authentication of a user, your program should fork and start a terminal window with the user's preferred shell. The parent process should just wait until the child exits and then show the `"login:"` prompt again.
- Before starting the terminal window, the child process should set the right real and effective user of the process, and the right real and effective group.

Try your program. You should also try to login with some other username/password, here is a list of users/passwords who has an account on your computer:

- donald/quack01

- daisy/quack02
- scrooge/quack03
- mickey/cheese01
- minnie/cheese02
- goofy/peanut01

Problem 9. *Why does your program only work when you login with your own username/password.*

Execute `"niceify ./mylogin"`. If your program has any other name, change the command as required. Run your program again and try to login as someone else. Does it work? After compiling you may need to run `niceify` again. In the terminal window you may check you identity using the id command.

Problem 10. *What does* `"niceify"` *do? Is it a good (secure) thing to have lying around in a system?*

You have finished **Part 1** of the lab, ask the lab assistant to check your program and your answers.

Part 2: File Access Schemes.

The file access scheme used in UNIX/Linux is not as flexible as the one used in e.g. Windows NT. For a regular file the permission are quite easy to understand but directory permissions are a little more difficult.

Create a subdirectory in you home directory and put a file **welcome.txt** with a short message in this subdirectory. Set the permission bits of the subdirectory so that the owner has **execute** access.

Problem 11. *Can you do the following with this permission:*

- *Make the subdirectory the current directory with* `cd`.
- *List the subdirectory.*
- *Display the contents of* ***welcome.txt****.*
- *Create a copy of* ***welcome.txt*** *in the subdirectory.*

Repeat the same experiments first with ***read*** *permission and then with* ***write*** *permission on the subdirectory.*

Problem 12. *Which flags need to be set in order for creation of new files to work? And for copying files?*

Check your answers.

Remove the **welcome.txt** file and the subdirectory you created.

Part 3: Password cracking on Linux - Hacking into a Computer.

Hacking into a computer is somewhat dubious to teach in a course, but if you are going to learn about computer security it is also important to know about computer insecurity. Don't run the

password cracking program we are going to use in this lab on any of the school's computers. Trying to find another users password is an offence and you will be expelled. Even listing the password file could be an offence.

As you saw in **Part 1**, every user with an account on your computer can read the `/etc/passwd` file. In most UNIX/Linux systems today the system administrator can prevent this by using so called *shadowed* passwords. With shadowed passwords the password field in `/etc/passwd` has just one character, an **x**, and the real encrypted password is stored in another file which is only readable by the superuser **root**. Ok, you think, but why can't you just make `/etc/passwd` only readable by root? All users home directory are, as you know, also stored in `/etc/passwd` and when you for instance write `"cd ~labuser/"` your shell (which has you as effective user) needs to read user `labuser`'s home directory from `/etc/passwd`. This functionality is standard in UNIX and can not be changed, so people came up with the *shadow* solution instead.

There are several ways to hack into a networked computer. You could try an anonymous ftp-login, and then try to extract user information. Sniffing the network for passwords sent in plaintext is a commonly used method, as you will learn in Lab 3. Another way is to try to get the password file of the system and mount a so called off-line attack. In an on-line attack you must be connected to the target machine throughout the attack, you don't need that in an off-line attack. If you somehow get hold of the target computer's `passwd` (or `shadow`) file, you can try to find a matching password on another computer without the target machine knowing about your activity. Of course, it might know that you read the password file by auditing the activity on the password file. Another very efficient method is the one you are going to explore here.

Preparatory assignment

- Read the documentation for the program *john the ripper*. You can find it at: `http://www.openwall.com/john/doc/`.

On the lab network there is a computer called `crackme.lab.local`. Some of the local users on your computer also have an account on `crackme`. Assume that some of those users have chosen the same password on your computer as they have on `crackme`. This is in fact not very unlikely.

- Copy the file `/usr/local/john-1.7.0.2.tar.gz` to your home directory. Unpack and extract the archive (`tar -xzvf john-1.7.0.2.tar.gz`).
- Compile the source by running `make clean linux-x86-mmx` in the `src` subdirectory.
- Issue `cd ../run` to find the executables.

Problem 13. *Use the built-in function to benchmark how fast passwords can be checked (see documentation for how to do this). Which one-way-function is fastest to brute force? Which is slowest?*

Copy the password file to the `run` subdirectory and start cracking the file.

Problem 14. *Gain access to the computer* `crackme.lab.local` *using telnet.*

You are now finished with Lab 2!

Appendix A, Compiling C Programs

This appendix is not mandatory to read if you know how to compile and link C programs and search the manual pages.

The C compiler you are going to use is GNU gcc. Gcc is not really the compiler but a "front-end" to a collection of compilers, (GNU Compiler Collection, gcc). The syntax of the gcc is:

```
patrike@elg-2>gcc [options] source1.c source2.c ...
```

The compiler names the output file `a.out` by default. You can specify the output name by passing a option `"-o outfile"` to the compiler. There are several other useful options.

- `"-g"`, compile with debug information. Necessary if you want to debug you program. A reasonable good debugger is the *Data Display Debugger*. You invoke it by the command `"ddd ./myprog"`.
- `"-l`*library*` "`, link your program with library *library*. You can list the `/usr/lib` and the `/usr/local/lib` directories to check which libraries you have installed on you system. If the manual tell you to link your program with the `crypt` library you include the option `"-lcrypt"`, or if you need the math library you include the option `"-lm"`. The standard library is always included. Most libraries have file name that starts with *lib*, e.g., `libxxxx.a`. But when you tell the compiler to link to this library, you give the option `"-lxxxx"`. The option `"-l"` replaces the `lib` in the file name.
- `"-Wall"`, compile with all warning message on. This is a good option to use, since this will catches assignment problems and warn you of type casts, which the compiler normally would not argue about.
- `"-O`*n*`"`, optimize your program. *n* is an integer from 0 to 3 where 0 is no optimization and 3 is full optimization. On some systems you can have higher level of optimization than 3.
- `"-c"`, only compile and not link. If you don't specify any output file, the compiler will use the same name as the source file, but change the extension to `.o`.

To compile the program *myprog.c* which uses functions from another file *myfuncs.c*, and somewhere in the program the `crypt(3)` function is used, you write:

```
patrike@elg-2>gcc -Wall -g -o myprog myprog.c  myfuncs.c -lcrypt
```

Observe that *myprog.c* and *myfuncs.c* can not both have a main function. You can read more about how to split your program into smaller files in [1].

Using the manual program

In the assignments you will need to use library routines which are probably not described in a standard textbook on C. Most of these functions are in the standard library, to which the compiler will link our program automatically. However, for some functions, you will have to read the manual pages to understand how they work and how to call then. The manual pages are accessed by running `"man`*item*`"` in a UNIX shell. This command will search the manual for *item*, which could be e.g. a program, a system command, a C function or a system file.

The manual is divided into sections. Some words are in many sections, e.g. if you write `"man passwd"` you will get information about the user program `passwd` (which is in manual section 1). If you write `"man 5 passwd"`, you will access section 5 in the manual, which has information about the file `/etc/passwd`. If you don't specify any section, the `man` program will search the sections in order and the first match is displayed. The different sections concerns different things, and to confuse the user even more, different UNIX systems have different sectioning systems. You can search the sections for a specific command by entering `"man -k`*command*`"`. In this guide we will refer to the sectioning system used in Debian GNU/Linux:

1. Executable programs or shell commands
2. System calls (functions provided by the kernel)
3. Library calls (functions within program libraries)
4. Special files (usually found in `/dev`)
5. File formats and conventions eg `/etc/passwd`
6. Games
7. Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7)
8. System administration commands (usually only for root)
9. Kernel routines [Non standard]

When we write `passwd(5)` we mean the file format of `/etc/passwd` and not the `passwd(1)` user program. You can read more about the `man` program by running `"man man"`.
Remember: If you wanna know, RTFM...

Crash course in C pointers and function arguments.

As you probably know, the memory in a standard computer is a contiguous entity of bytes. Each byte has a unique address. So if you declare a variable

```
char c;
```

You will reserve space somewhere in memory for a char type. A char is the C-way of declaring a byte. The address of `c` can be determined by using the unitary operator `&` on `c`. So if you write `&c` you mean the *address* of `c`. To store the address of `c` in some other variable you need to declare a *pointer* variable and then assign its value. You do this by

```
char c;
char *p;  /* pointer to a char */
...
c=3;    /* assign c the value 3 */
p=&c;   /* the value of p is the address of c */
```

Now, if you want to know the value of `c`, you have two different possibilities of asking:

- *what is the value of c?*
- *what is the value of the char that p is pointing to?*

The second question is written in C as `*p`. The operator `*`, means dereferencing; when applied to a pointer `p` , it tells you the value of the item `p` is pointing at.

```
 0: #include <stdio.h> /* header file for standard in- and output */
 1:
 2: int main() {
 3:
 4: char c;   /* declare a char */
 5: char *p;  /* decalre a pointer to char */
 6:
 7: c='A';   /* Assign hex value 0x41='A' to c */
 8: p=&c;    /* p is pointing to the address of c */
```

```
 9:  /* print c as integer and as ASCII character */
10: printf("Test 1: %d %c\n",c,c);
11: /* Print the value p is pointing at. Integer value and ASCII */
12: printf("Test 2: %d %c\n",*p,*p);
13: return(0);
14: }
```

When assigning values to a variable of type char, you can either specify it as an integer in the range $0 \ldots 255$ or as an ASCII character, like we do in this example.

Array equals Pointer

An array is a declaration of "*more than 1 items in a row*". For example,

```
char a[10];
```

reserves a row of 10 chars in memory, and `a` is a constant *pointer* to the first element. To access the value of the first element you would write `*a` and to access the value of the second element you would write `*(a+1)`. Here you see that you can do arithmetics on pointers. *One important thing to remember is that the elements in* `a` *have index* $0 \ldots 9$. A short hand notation for `*(a+`n`)` is `a[`n`]`, but you should always remember that `a` really is a pointer. If you would try to access `*(a+10)` you'll be reading outside the reserved memory for `a`. This is a very common programming error, since C does not provide any runtime checking of your pointer arithmetics. Let's look at an example:

```
char a[10];  /* declare an array of 10 bytes, where a points to the first element */
char *p;     /* declare a pointer to char */
...
*(a)=2;      /* same as a[0]=2; */
*(a+1)=3;    /* same as a[1]=3; */
p=a;         /* p points to the first element in a, i.e. *p==2;  */
p++;         /* you can increase a pointer and now *p==3; */
             /* p points to the second element in a */
a++;         /* ERROR, you can't increase a, it's a constant pointer */
p=p+9;       /* p points beyond the array a */
p=a+9;       /* now p points to the last element in a, *p==a[9] */
...
```

As you see in this example, you can not change the address to which `a` is pointing. When you declare

```
char a[10];
```

you make `a` a constant pointer which can not be altered.

You can have pointers to other types than char as well. An array of integers and a pointer to an integer is defined as

```
int ia[10];
int *ip;
```

On a Pentium machine, an integer is 4 bytes and the compiler keeps track of the size of the item your pointer is pointing to. Using the above declaration, we look at another example:

```
ip=ia;          /* ip points to same address as ia */
*(ip)=2;        /* same as ia[0]=2; */
*(ip+1)=3;      /* same as ia[1]=3;  */
```

So when we write `(ip+1)` the compiler knows that it should increase `ip` with 4 to point at the next item, whereas `p+1` in the previous example meant to increase the address by 1, since a char is only one byte.

Multidimensional arrays can also be declared:

```
char a[10][20]; /* reserves memory for 10*20=200 char in memory */
                /* and a is a pointer to the first element      */
char *ptr[10];  /* reserves memory for 10 pointers to char      */
```

Observe that you have only reserved memory for the pointers in the second declaration. To use e.g. `*ptr[1]` for data storage you must allocate memory first.

```
ptr[1]=(char *)malloc(20); /* reserve a block of 20 chars in memory */
                           /* and put the address in ptr[1] */
*ptr[1]='E';    /* the first char in the newly malloc:ed string */
*(ptr[1]+1)='F';   /* second char...*/
*(ptr[1]+2)='D';   /* third char ...*/
*(ptr[1]+3)='\0'; /* remember to end strings with a NULL char */

printf("%s\n",ptr[1]); /* print the string ptr[1] */
```

`ptr[1]` is a pointer to a (with malloc) reserved area of size 20 chars, so `ptr[1]+1` must be the next address in memory, i.e. the address to the second char in our string. Finally, the contents of that address `*(ptr[1]+1)` is assigned the value `'F'`.

Call by reference/value.

Consider the following code with line number to the left:

```
 0: /* Version 1, the erroneous way */
 1: #include <stdio.h>  /* include header declarations for */
 2:                     /* standard input and output functions like printf() */
 3: void square(int x) {
 4:    x=x*x;       /* square x and return */
 5: }
 6:
 7: int main() {
 8: int a;
 9:
10:   a=2;
11:   square(a);   /* call function to square a */
12:   printf("%d\n",a);  /* print a */
13: }
```

This program will **not** print 4 as its result. The problem here is that it is only the value of `a` that is passed on to the function `square`. When you update the value of `x=x*x` inside the function it will not reflect on the value of `a` after the function. This is called *call by value*. What you must

do to make it work properly is to provide the address of `a` to the function, so that `square` can update the contents of the address to `a`. The proper program would be:

```
 0: /* Version 2, the correct way */
 1: #include <stdio.h>
 2:
 3: void square(int *x) {  /* void means that the function gives no return value */
 4:    *x=(*x)*(*x);       /* square the value that x is pointing at and put     */
 5: }                      /* the result at the same address */
 6:
 7: int main() {
 8: int a;
 9:
10:   a=2;
11:   square(&a);   /* square the contents of the memory address where a is located */
12:   printf("%d\n",a);  /* print a */
13: }
```

So this will print 4 as its result. What we have done here is to call `square` with a address reference to `a`, and it is called *call by reference*. Of course, the easiest way of implementing the `square` function would be to let the function return the result:

```
 0: /* Version 3, the normal way */
 1: #include <stdio.h>
 2:
 3: int square(int x) {  /* now the function returns an integer */
 4:    return(x*x);       /* square the value of x and return it */
 5: }
 6:
 7: int main() {
 8: int a;
 9:
10:   a=2;
11:   a=square(a);   /* square value of a and put the result back into a */
12:   printf("%d\n",a);  /* print a */
13: }
```

But sometimes you need to change the value of several arguments and since functions only return *one* value, you must use the *call by reference* method.

If you have a pointer as argument, and your subfunction will change the address to which the pointer is pointing, you must provide the address of the pointer, and the function will take as argument a "pointer to a pointer". This might look like:

```
 1: #include <stdio.h>
 2:
 3: void inc_pointer(int **ipointer) {
 4:  /*  int **ipointer means an address to an address to an integer */
 5: (*ipointer)++;    /* increase the address to which *ipointer points */
 6:}
 7:
 8: int main() {
 9: int ia[10];
```

```
10: int *ip;
11:
12: ia[0]=2; ia[1]=3;   /* put some values in the integer array */
13: ip=ia;              /* ip points to the same address as ia */
14: inc_pointer(&ip);   /* after this, ip point to to the second item in ia */
15: printf("%d\n",*ip);  /* print the value 3 */
16: }
```

A more thorough description on pointers and arrays can be found in Chapter 5 of [1]

Appendix B, List of files available for download.

This is a short description of the programs that you can download from the web.

- `lab2.pdf` : This lab guide.
- `userinfo.c` : Demo program that prints information about users from /etc/passwd.
- `pwdblib.c` and `pwdblib.h` : Routines to access and update the local password file.
- `pwdb_showuser.c` and `pwdb_updtuser.c` : Demo programs that uses `pwdblib.c`.
- `openshell_demo.c` : Demo program on how to fork a process and open a terminal window.
- `pwfile` : Example of a local password file to be used with pwdblib.c.
- `lab2kit.zip`: All above C files and this pdf lab guide in a zipped file.

References

[1] B.W. Kernighan, D. M. Ritchie, *The C Programming Language*, Second edition, Prentice Hall Software Series, Prentice Hall 1988, ISBN 0-13-115817-1.

[2] S. Loosemore *GNU C Library Reference Manual*, Edition 0.07 DRAFT, Free Software Foundation, URL:http://www.gnu.org/manual/glibc-2.0.6/