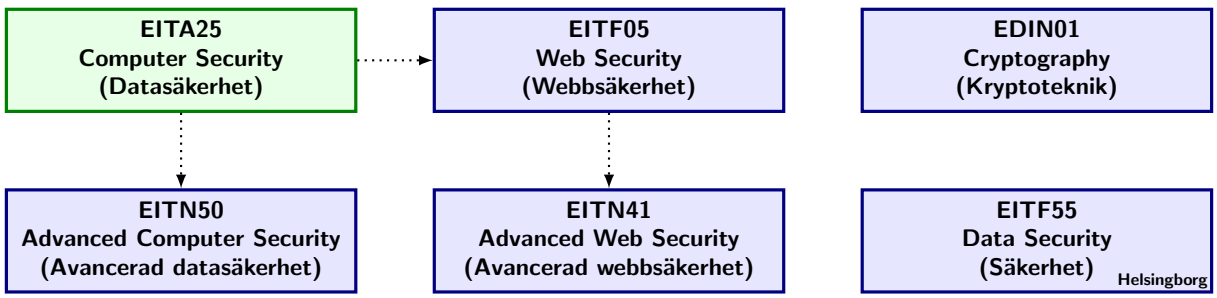
Computer Security 2020

Lab 2: UNIX/Linux security: Login, file access schemes, and shared directories

- This lab will be done in groups of 2 people.
- There are preparatory assignments for this lab, read through the complete lab guide carefully, and bring your written answers to the lab.
- During the lab, write down answers to all problems on a sheet of paper so your work can be approved.

Learning goals:

- Learn how the login program works in Linux.
- Understand the file access scheme in *nix-systems.
- See how you can set up a shared directory on Linux.



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. *You must bring your preparatory assignments to the lab, and remember that you do not have access to the Internet!* A USB-stick with your `.c`-files is probably the best way to accomplish this. 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 files. These files can be found on the course webpage. Check the webpage and download the files before you start preparing. A short description of each available file is also listed in Appendix A.

Note that you will not have any internet access during the lab, so come prepared. You may bring as many books and printed materials as you can carry. 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. Alternatively, if you feel confident in the availability of eduroam, you may bring your own laptop, smartphone, or tablet to get Internet access.

Introduction

The computers in this lab will run Linux. 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 B 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 CentOS, but should work on at least most Linux distributions, even though different flavors of UNIX have slightly different commands. All files that you should download and study will be available on your computer in the directory `/mnt/server.local.lab/eita25/lab2` when you do your lab.

> **Preparatory assignment 1**
>
> - Read about Linux security, for example in the online chapter from the course book, or the lecture slides for the Linux lecture.
> - Ensure that you know what do to when we ask "Read the manual pages for `login(1)`". Read Section B.0.1 if you are unsure.
> - Read the manual pages for `login(1)` and `passwd(5)`. Make sure you understand the format of `/etc/passwd`.
> - *Important!* You will need to use the Linux command-line during some parts of this lab. Make sure you know how to perform operations such as: changing directory, listing directory contents, running programs, change permissions using `chmod`, etc. at the command-line. A good resource is the booklet from the UNIX introduction of the first year (Datorstugehftet). Bring it, or study the PDF version at DDG's website `http://www.ddg.lth.se/perf/unix/`

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 all parts, and as you proceed with the lab, 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`. 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`.

In modern versions *nix-operating systems, `/etc/passwd` will by default not contain any hashes (more on this later). This includes LTH's Linux computers and your own (current century) Linux installation at home. To be able to test your solutions to the preparatory assignment on any computer, your program will use a local file in the same directory as your program, instead of using the system-wide password file. This local file is called `pwfile`. The file is similar in format to `/etc/passwd`, but you will have to use other functions to manipulate and access it, so that you do not alter or read the system file.

Preparatory assignment 2

- Look at the contents of the supplied `pwfile`, and compare it with the format used in the real `/etc/passwd`. What are the differences? The format of the real `/etc/passwd` is described in the lecture slides, and also in the man page `passwd(5)`. The format of the `pwfile` can be found by looking in `pwfile`, and by checking the definition of `struct pwdb_passwd` in the downloaded file `pwdblib.h`.

For your program to use the local `pwfile` instead, you must prefix some function calls with `pwdb_`. In this assignment, you may for example use `pwdb_getpwnam` instead of `getpwnam`. The `pwdb_getpwnam`-function should work the same way as `getpwnam`, so you may use the man page for `getpwnam(3)` to see how the function should be used in your program.

Preparatory assignment 3

- Study the downloaded program `userinfo.c`. Copy it to a new file called `mylogin.c`. Rewrite it so that it mimics the system login procedure, which for example includes that it should not echo the password as you type it on the screen. The program should check the password entered by the user with the corresponding entry in the `pwfile`-file. Useful library functions might include `pwdb_getpwnam`, `getpass(3)`, `crypt(3)` and `strcmp(3)`. You may ignore warnings that `getpass(3)` is deprecated. If the passwords match, the program should write something like *"User authenticated successfully"* and terminate. If the password is wrong, or the username invalid, it should respond *"Unknown user or incorrect password."* and start over with the `"login:"` prompt.

Problem 2 *Why do we use the same error message for both "Invalid user" and "Wrong password"? Would it be a problem to print two different error messages instead?*

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, and to also compile the* `pwdblib.c`*-file*[a]*. Does it work as you*

expect? See below for some username/passwords to test with.

[a] A suitable command for compiling may be: `gcc -o mylogin mylogin.c pwdblib.c -lcrypt`
If you are using Mac OS X, leave out the `-lcrypt` flag.

The `pwfile`-file contains a couple of accounts. Some of them are listed below, you can use them to test your programs.

- donald/quack01
- scrooge/quack03
- minnie/cheese02

In the first part you have been accessing `pwfile` in the same way as you would have accessed `/etc/passwd`. From now on, you are going to use the extra fields in our `pwfile` to extend the functionality of the program.

Preparatory assignment 4

- Make a copy of your original `mylogin.c`-file, and implement the following new features in the new file. For an example of how you update the `pwfile`, look at the provided example in `update_user.c`
  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 or her 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? (You will need to test this on Linux, not Mac OS X!). Why is it bad if we can terminate the login program?*

Secure your program against *Control-C*, such that your program doesn't exit when the combination is pressed. Check the man pages for `signal(2)` and `signal(7)`. You can read more about signals in Chapter 24 in [2]. Is there any other signal you should protect your program against? If you need to kill your program after blocking Ctrl-C, write `killall -9 mylogin` in another terminal.

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 5**
>
> - 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 the chapter on processes in the *GNU C Library Reference Manual* [2]. You can download the complete manual or the relevant chapters from the lab homepage or find it in `/mnt/server.local.lab/eita25/lab2`.

> **Problem 7** *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. When the shell window has opened, try to write the command `id`. Which user and group id is logged in?

> **Problem 8** *Why doesn't your program switch to the correct user and group?*

> **Problem 9** *Run `ls -l` and write down the permissions and owner of `mylogin`.*

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 need to run `niceify` again. In the terminal window you may check you identity using the `id` or `whoami` command. You only need to ensure the `uid` and `gid` are correct, but an interested student may also fix the supplementary group list (called `groups`). *Hint:* If you see the message `/home/localhost/eita25/.profile: Permission denied` you have probably succeeded. Why do you get this error message? *Hint 2:* If only the `uid` changes, but not the `gid`, think about the order of your `setuid` and `setgid` calls. Why does the order matter?

> **Problem 10** *What does* `niceify` *do? Is it a secure thing to have lying around in a system?* Hint: *Check the permissions and owner again—compare it to your answer from Problem 9.*

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. modern version of Windows. 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 *only*. You should do this using the terminal, since the graphical interface may alter the permissions in unexpected ways.

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.*

Problem 12 *Repeat the experiments from the question above for the following three cases. If you want to, you can fill in the results in the fancy table below:*

- *The subdirectory has* ***read*** *permission* ***only***.
- *The subdirectory has both* ***read*** *and* ***execute*** *permissions.*
- *The subdirectory has* ***read, write****, and* ***execute*** *permissions.*

Subdir permissions	*Operations*			
	current directory	*list contents*	*display* `welcome`	*copy*
`r--`	☐	☐	☐	☐
`r-x`	☐	☐	☐	☐
`rwx`	☐	☐	☐	☐

Problem 13 *Which flags need to be set on a directory in order to create new files?*

Remove the `welcome.txt` file and the subdirectory you created. Check your answers and show them to your teaching assistant.

Part 3: Sharing files on a Linux system

Now, let's look at how file sharing on a shared host can be accomplished in Linux systems. Start by using ssh to login to the computer `crackme.local.lab`. Use your computer name as username. The password is `Kanejbytas123` for all accounts. Thus the command is:

```
ssh linaXX@crackme.local.lab
```

Problem 14 *Which groups does your linaXX account belong to?*

Go to the directory `/home/shared`. Start by creating a directory for your own computer linaXX.

Problem 15 *Who is the owner of the directory? Which group does the directory belong to?*

We want to share this directory such that every user with the same colour as yourself can both read existing files, and create new files, in your newly created `linaXX` directory. Other users should not be able to access the files, nor see the content of the directory.

Problem 16 *Describe what owner, group, and permissions your directory should have to accomplish this, and set them on your directory.*

Enter your newly created directory and create a file with some contents[1]. Ask a neighbour to go into your directory and ensure that the file can be read. Also ask your neighbour to create a new file in your directory as well. If you don't want to talk to your neighbour for some reason[2], you may instead create an extra ssh-connection, using the username of your neighbour.

Problem 17 *Can you modify the file your neighbour created? Why or why not?*

Now, set the setgid-bit on your directory. Recall from the lectures that this can be done using the command:

```
chmod g+s path/to/directory
```

Create a new file in your directory and check the permission, owner, and group of the newly created file.

Problem 18 *What's different from before? What does the setgid bit do?*

Finally, let's look at deletion of files. Go to the directory `/home/shared/everyone`. Look at the permissions, owner, and group of all the files using `ls -l`.

Problem 19 *Try to delete **one** of the `hejXX` files. Can you delete it? Why or why not?*

Now, try to delete the file `/home/shared/annoying.txt` instead.

Problem 20 *Why can't you delete the file? What is different compared to the file in the previous question?* Hint: *Compare permissions of both the files **and** the files' parent directories.*

You are now finished with lab 2! Delete the `linaXX` directory you created on `crackme`, and show your answers to the lab assistant.

[1] Use `nano`, `vim`, or just write `echo hejsan > fil.txt` to create a file named `fil.txt` with the contents `hejsan`.
[2] Talking to other people is actually nice. I can wholeheartedly recommend anyone to try it once in a while!

A List of files available for download.

This is a short description of the files in the `kit2019.tar.gz` archive that you can download from the web.

- `userinfo.c` : Demo program that prints information about users from `pwfile`.
- `pwdblib.c` and `pwdblib.h` : Routines to access and update the extended version of the password file.
- `update_user.c` example program that shows how to update a user's information.
- `openshell_demo.c` : Demo program on how to fork a process and open a terminal window.
- `pwfile` : Example of a local extended password file to be used with `pwdblib.c`.

B 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:

```
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:

```
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].

B.0.1 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 the file `/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...

B.1 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.

```
 1: /* header file for standard in- and output */
 2: #include <stdio.h>
 3:
 4: int main() {
 5:
 6: char c;   /* declare a char */
 7: char *p;  /* decalre a pointer to char */
 8:
 9: c='A';   /* Assign hex value 0x41='A' to c */
10: p=&c;    /* p is pointing to the address of c */
11: /* print c as integer and as ASCII character */
12: printf("Test 1: %d %c\n",c,c);
13: /* Print the value p is pointing at.
14:  * Integer value and ASCII */
15: printf("Test 2: %d %c\n",*p,*p);
16: return(0);
17: }
```

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.

B.1.1 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:

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

```
 * it's a constant pointer */
a++;
/* p points beyond the array a */
p=p+9;
/* now p points to the last element in a, *p==a[9] */
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 32-bit Intel 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:

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

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.

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

B.1.2 Call by reference/value.

Consider the following code with line number to the left:

```
 0: /* Version 1, the erroneous way */
 1: #include <stdio.h>
 2:
 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: /* void means no return value */
 3: void square(int *x) {
 4:    /* square the value that x is pointing at */
 5:    /* and put the result at the same address */
 6:    *x=(*x)*(*x);
 7: }
 8:
 9: int main() {
10:   int a;
11:   a=2;
12:   /* square the contents of the memory address
13:    * where a is located */
14:   square(&a);
15:   printf("%d\n",a);  /* print a */
16: }
```

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) {  /* returns an integer */
 4:    return(x*x);      /* square and return */
 5: }
 6:
 7: int main() {
 8:   int a;
```

```
 9:
10:   a=2;
11:   a=square(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: /* int **ipointer means an address to an
 4:  * address to an integer */
 5: void inc_pointer(int **ipointer) {
 6:   /* increase the address to
 7:    * which *ipointer points */
 8:   (*ipointer)++;
 9: }
10:
11: int main() {
12:   int ia[10];
13:   int *ip;
14:
15:   /* put some values in the integer array */
16:   ia[0]=2; ia[1]=3;
17:   /* ip points to the same address as ia */
18:   ip=ia;
19:   /* after this, ip point to to the second
20:    * item in ia */
21:   inc_pointer(&ip);
22:   printf("%d\n",*ip);  /* print the value 3 */
23: }
```

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

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*, Free Software Foundation, `http://www.gnu.org/software/libc/manual/pdf/libc.pdf`