Version Wed, Feb 21st, 2017

Week 02

Deadline: Saturday, February 25th 2017

Privacy Matters, Encryption and Digital Signature using GnuPG

1. Login to your badak account
2. create new directory named "`work`" right where you first login and directory named "`work02`" inside "`work`" directory

```
$ mkdir work
$ cd work/
$ mkdir work02
```

3. change your current directory to `work02`
4. you must copy 2 files for this week's task (you can download it on SCELE week02 forum) and put it in "`work02`" folder
   a) 00-toc.txt
      this is the "command-oriented-version" file of this tutorial. Save it into a file named "`00-toc.txt`"
   b) 01-public-osteam.txt
      this is osteam's public key, for. Save it into a file named "`01-public-osteam.txt`"

   You can copy these two file by learning about **tunnels**, or using the "(win)SCP" steps ahead.
5. to copy files in step 4, you can learn how to do it by reading tutorial TutorialWinscp.pdf
6. Check whether the 2 files copied safely by using ls-al command.

```
$ ls -al | tee 02-ls-al.txt
```

7. we will use *Public-key Cryptography* to secure your works.
   what is that? You can think it as a lock and key method.
   "At the first, everyone has a lock (public key) and a key (private key). And then when you send a letter to someone, to make sure the letter cannot be read by someone else, you lock (encrypt) it using a 'new' lock that created based on your lock and recipient's lock. This 'new' lock is so secure that only recipient can open it using their key (even you cannot open it if you not define it in creating process)"
   You can see more info and more accurate example in wikipedia

8. in this class, you will (always) encrypt your works so that only you, teaching-assistant and lecturers can see it. so, in this case the *"recipient"* is us, osteam
9. first, let's create your own lock and key. We will use GnuPG tool to do that.

```
$ gpg --gen-key
```

create your GnuPG key using these details:

a) RSA & RSA encryption method, 4096-bit long

b) 6 months' key validity

c) Name: your complete name

d) Email: your UI email

e) Comment: any comment (example: "studying OS is ez!")

f) Passphrase is the same as password but for public key

10. if success, your key should be listed. Check it and put it into file named "`03-list-keys1.txt`"

```
$ gpg --list-keys | tee 03-list-keys1.txt
```

Inside the file, should be like this

```
/home/fasilkom/mahasiswa/i/ibad.rahadian/.gnupg/pubring.gpg
------------------------------------------------------------
pub   4096R/2B615FE5 2017-02-15 [expires: 2017-02-21]
uid                  Ibad Rahadian Saladdin (OS MANCAY)
<ibad.rahadian@ui.ac.id>
sub   4096R/3BEF4F02 2017-02-15 [expires: 2017-02-21]
```

*pub, is for the primary to be used for signing.

*sub is for subkeys, used to encryption.

11. before we start encrypting for this class, import osteam's public key using command "`gpg --import <key_file>`"

```
$ gpg --import 01-public-osteam.txt
```

12. check it and put it into file named "`04-list-keys2.txt`"

```
$ gpg --list-keys | tee 04-list-keys2.txt
```

13. export your public key to file named "`mypublickey1.txt`"

```
$ gpg --export --armor > mypublickey1.txt
```

*to make sure, you can open "mypublickey#.txt". It should be filled with random strings.

14. move the file "mypublickey1.txt" to your GitHub directory on key folder (to make it easier to find) using command "mv <file name> <filepath>/<filename>"

```
$ cp mypublickey1.txt ~/os171/key/mypublickey1.txt
```

*Note that if you make new GnuPG key, you need to move the file named as "mypublickey#.txt" with # as the number you have made the file.

Intro to C

15. now before we continue our encryption learning session, we will learn about C programming.
16. in C, there are two kinds of files. The *.h and *.c
17. the *.h file known as *"header file"*. Simply put, you can think it as *interface* as in *java*. And the *.c is the main file where you put your implementation code.
18. Let's write our first C program. Create a file named hello.c and edit it with the following code:

```
#include<stdio.h>
int main(void) {
	printf("hello world\n");
	return 0;
}
```

* you can think "#include" as *import* as in *java*

You can easily create new files with your favorite text editor command "vi hello.c" or "nano hello.c"

19. just like in *java*, you must always compile it first before you can run it. To compile a *.c program, use command "gcc <file> [-o <output_name>]"

* note, <output_name> argument is optional, if not specified it will use name "a.out" (default)

```
$ gcc hello.c -o hello
```

now try to run it

```
$ ./hello
```

the function `printf` is similar to `print` in *java*, which will print the string inside the parameter to the console. In this case:

```
$ hello world
$
```

20. now for another example, we will learn a function from C called, `getpid()` and `getppid()`. Create a file named `status.c` and edit it with the following code:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

void main() {
   printf("PID[%d] (PPID[%d])\n", getpid(), getppid());
   sleep(1);
}
```

21. we will use the header `stdio, sys/types,` and `unistd` (you can always ask uncle G about those header). Compile the program with the name `status` and run it. The function `getpid()` or "get process ID" will get the **process ID** of the current process, and `getppid()` or "get parent's process ID" will get the **process ID of the current process's parent process**. This may sound confusing, but you'll learn about this later on in lesson **process & thread**.
22. now for the last example, we will learn about loop in C. Create a file named `loop.c` and edit it with the following code:

```
/*
 * (c) 2013-2016 Rahmat M. Samik-Ibrahim
 * This is free software.
 * REV00 Tue Sep 20 16:43:44 WIB 2016
 * START 2013
 */

#define LINETXT "==========\n"
#define OLOOPTXT "OL [%4.4d]\n"
#define OLOOP  3
#define ILOOP  4

#include <stdio.h>
void main() {
   int ii, jj;
   printf(LINETXT);
   for (ii=0; ii<OLOOP; ii++) {
      printf(OLOOPTXT, ii);
      for (jj=0; jj<ILOOP; jj++) {
         printf("    IL[%d]\n",jj);
      }
   }
```

```
   printf(LINETXT);
}
```

23. compile the program with name `loop` and run it. The code above is an example of loop in C. Pretty similar to *java*, no? The `#define` is used to make a constant variable (unchanging variable) and can be used throughout the code. The syntax `for` is similar to what you can find in *java*.
24. now for the exercise, create a program in C called `exercise.c` and edit it with the following template:

```
/*
 * Name: [YOUR_NAME]
 * NPM: [YOUR_NPM]
 * Class: [YOUR_CLASS]
 * Comment: [YOUR_COMMENT]
 */

#define LOOP //define me!

#include <stdio.h>

void main() {
   int input = //define me!

   // TO DO implement me!!
}
```

25. replace the [YOUR_NAME], [YOUR_NPM], [YOUR_CLASS] with the valid info, and [YOUR_COMMENT] with your comment about this lab (e.g. “What is this?” without quote sign)
26. then, create a program that can calculate the product of variable `input` and constant `LOOP` using **loop operation only (for or while), operator “*” is not allowed** and then print with format “`[input]` times `[LOOP]` equals `[THE_PRODUCT]`“ (`[THE_PRODUCT]` can be anything, as long as it holds the product of multiplication). For example, when the `input` is 5, `LOOP` is 4, thus it will print “5 times 4 equals 20”. For submission, you can define the value of `[input]` and `[LOOP]` to **any integer n that satisfies n > 0** so it can be used to prove that your program is correct (correctors will check your source code whether it matches the program’s output)
27. compile the program with name `exercise` and run it
28. congratulation, now you already know a small portion of how to program in C

Back to Privacy Matters, Encryption and Digital Signature using GnuPG

29. now let's get back to our encryption learning session: sign your works so the other know it truly your works. First, we generate hash of all your works. So, instead we sign every file, we just need to sign the hash. We will use *SHA-1 algorithm* for now. (ask *uncle G* for more information about *hashing* and *SHA-1*). Execute the following commands

```
$ sha1sum * > SHA1SUM
$ sha1sum -c SHA1SUM
```

the first command is for creating `SHA1SUM` to all files in current directory **(`work02`)** and save it into a file named "`SHA1SUM`"

the second command is for verify the hash of your files. The "`OK`" output means there is no modification in your current files (try to modify some of your files and then verify it). So, **if you modify your files, you must generate the new hash of it.**

30. To sign a file, use command "`gpg --sign --armor --detach <file>`". The "`--sign --armor --detach`" argument means *"create detached signature with ASCII armored (human readable) output"*

```
$ gpg --sign --armor --detach SHA1SUM
```

after this, there should be a file called `SHA1SUM.asc`, the signature file. This file contains hash and information of the one who signing the file (again, checked by public key)

31. try to verify your signature. Use command "`gpg -verify <signature_file> [<file>]`". The "`<file>`" argument is optional. If not provided, it will try to use `<signature_file>` without "`.asc`" suffix (e.g `test.asc → test`)

```
$ gpg --verify SHA1SUM.asc
```

so, as long as the signature is good, we can ensure that the files in this folder **(work02)** is not being modified from the last time of your works

32. create a tar ball. Tar is a way to create an archive file. They will be a new file called "`work02.tbj`". You can ask uncle G for more information.

```
$ cd ..
$ tar cvfj work02.tbj work02/
```

*Useful info: to untar the .tbj file, you can use command "`tar xvfj <file>`"

33. now you can start encrypting your files. To encrypt a file, use command "`gpg --encrypt [--recipient <recipient_identifier>] <file>`".

    The "`--recipient <recipient_identifier>`" argument define the one who can decrypt the file. You can define n many recipients.

    *Note you must import the recipient's public key before.

    encrypt the tar file

```
$ gpg --output work02.tbj.gpg --encrypt --recipient OSTEAM --recipient your@email.com work02.tbj
```

    *Use the same email as your Email input on GnuPG key generator.

    **Usefull info: after this, there should be a file called `<file>.gpg`. That is the encrypted version of your `<file>` (try to read the content if you can). That file only can be decrypted using receipent's private key (in this case either use osteam's or yours). To decrypt a file, use command "`gpg <file>`"

```
$ gpg <file>.gpg
```

34. copy the file to your github account, under the file week02/

```
$ cp work02.tbj.gpg ~/os171/week02/work02.tbj.gpg
```

35. change your directory to `~/os171/key/`
36. check whether there is a file named "`mypublickey1.txt`" if you don't find it, do the file moving once more from the "`work02`" directory
37. change your directory to "`~/os171/week02/`"
38. remove file named "`dummy`"
39. check whether there is a file named "`work02.tbj.gpg`" if you don't find it, do the copy once more from the "`work02`" directory
40. push the change to GitHub server. You can see GitHub tutorial in week01 forum
41. done

Review your Work

Dont forget to check your files/folders. After this lab, your current `os171` folder should looks like:

```
os171
	key
		mypublickey#.txt
	log
		log01.txt
		log02.txt
	SandBox
		<some_random_name>
	week00
		laporan.txt
	week01
		lab01.txt
		laporan.txt
		myExpectation.txt
		what-time-script.sh
	week02
		work02.tbj.gpg
			*work02
				*00-toc.txt
				*01-public-osteam.txt
				*02-ls-al.txt
				*03-list-keys1.txt
				*04-list-keys2.txt
				*hello.c
				*hello
				*status.c
				*status
				*loop.c
				*loop
				*exercise.c
				*exercise
				*SHA1SUM
				*SHA1SUM.asc
	week03
		dummy
```

```
week04
      dummy
week05
      dummy
week06
      dummy
week07
      dummy
week08
      dummy
week09
      dummy
week10
      dummy
xtra
      dummy
```

keep in mind for every files/folders with wrong name, you will get penalty point.

*means file that should be inside the archived file.