Source: *Stable Diffusion AI*

Laboratory 4. Cryptography Applications (II)

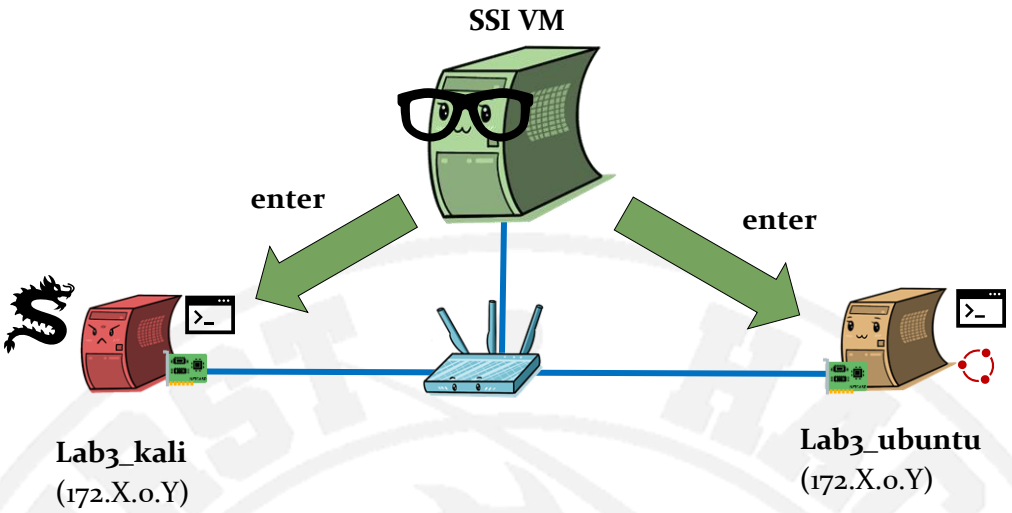
DEPARTMENT OF COMPUTER SCIENCE. UNIVERSITY OF OVIEDO

Computer Security | 2022 – 2023 (V2.7 *"S-81 Isaac Peral"*)

CONTENT

BLOCK 1: GNUPG MANAGEMENT

NOTE: This laboratory uses the exact same infrastructure than the previous one. This drawing reminds you how it was:



Generate a GPG key pair (public – private)

Practical application: *You can prepare a PKI infrastructure in your user account so now you can cipher and decipher files sent to / coming from known places / users*

In the previous laboratory we used `gpg` (`sudo apt install gnupg`) to practice symmetric-key encryption to encrypt files. Now we are going to review asymmetric encryption / digital signing to complement it. **We can use the same infrastructure as in the previous lab**.

The first thing we must do is to **generate a key pair for each user that will take part in the communication**. To generate a public/private key pair, follow this procedure:

- **GPG** needs entropy to generate truly random numbers that will be used to generate the keys. To increase the randomness, and to make the process shorter, we advise you to **start typing random stuff in the keyboard** when the utility prompts you to do it, so enough entropy will be generated quickly, and the key can be produced faster. Another option to increase entropy is to open a shell window into your VM and type this `sudo dd if=/dev/sda of=/dev/zero`, letting it run until the keys are generated.
- **We recommend you to begin with the *Kali* machine**, as the *Ubuntu* one is the only that has the proper software to accept keys you send via `scp`. However, as you can easily share contents between containers using the shared folders, is up to you!
- Run the following command to start key generation: `gpg --gen-key`
- We are now asked to provide an identity for the key holder. **We must take this step very seriously** as these keys will be used to identify each user, so we must provide a meaningful name. We recommend you to use `<your UO>ubuntu` in the *Ubuntu* container and `<your UO>kali` in the *Kali* container in the infrastructure. The user identities you use with certificates have not to be the same as your username, as they are independent even if the generated certificates are stored in the current *Ubuntu* user home folder. Provided email can be fake.
- Once the parameters are provided, we press "O" (OK) to begin key generation
- The generation process will need a password to store private keys in a secure way. Provide a password you can remember, as you will need it multiple times later. **Use good password generation practices for this on real applications!** 😊

Expected results: This activity finishes when you successfully generate a key pair for a user

Manage a Keyring

Practical application: *You need to consult or manage your current key pairs*

With GPG comes the concept of **keyrings**. **Every local user has their own public keyring and a private keyring that stores the keys that it generates or imports from other users**. The keys we have just generated are now on the respective user keyring, which can be managed using these commands:

- To view the keys on your private (secret) keyring, run: `gpg --list-keys` or `gpg --list-public-keys` for public ones
- If we have multiple public keys belonging to different users (as we will see in the next section), we can view the details of one or more specific public keys using the username provided upon key creation: `gpg --list-public-keys <username>` .The same can be done for stored private keys: `gpg --list-secret-keys <username>`

Expected results: This activity finishes when your generated public and private key can be correctly listed in your keyring.

Letting others know your public key

As said in the theory topics, your public key will be used by other users when they want to encrypt data that only you can read, as only your private key can be used to decrypt that data. **This process requires you to publish / export / send your public key** so anyone interested in sending you ciphered information can access to it. Therefore, you can give or email it to them, place it online, or store your key on a *keyserver* for others to retrieve. The procedure to follow is this

- To use one of the mentioned methods to send the public key, you must export it using the `--export` option. To export it in a manageable format we will use the `--armor` option of the previous lab and the username we provided to identify the public key we want to export. Do: `gpg --armor --export <username> > <username>_public_key.asc`, and please replace `<username>` by the appropriate user identity you used when the keys were generated, depending in the container you work with. **Please pay attention that there is a `>` character in the middle of the command!** 😊).
- Once you do this, the `<username>_public_key.asc` file will contain your public key. We recommend you to examine the contents of this file to understand it better.
- Other communication parties that want to use your public key to encrypt data need to import it into their own public keyring by using the `--import` option with the exported key file. Send the key to the Ubuntu container in the lab and import it. Once you do that in the other container, list the public keys of this second user and examine their contents.
- **NOTE**: If you have previously connected from the *Kali* container to the *Ubuntu* container, it is highly probable that `scp` throws an error telling you that the host is fake! ☹️ . This is normal; if you regenerated the lab since a previous execution the containers are indeed different, and in real life this could mean that somebody is impersonating the destination. Remove the previously stored remote site data with `ssh-keygen -f "/<current user id>/.ssh/known_hosts" -R "<IP of the remote container>"` and try to connect again. You can also use the shared directories of the containers to transfer files.

Expected results: This activity will be finished when you export a key of a user and import it into another user keyring.

Validate imported public keys by signing

When public keys of other users are imported to a keyring, **they should be also validated** by verifying the fingerprint using a secondary communication channel (e.g., over the phone). When the key is verified as correct, you should sign it to avoid future warnings. For example, to sign the imported `redondo` public key, you need to use the following procedure

- Edit the imported key: `gpg --edit-key redondo`
- This will bring you to a `gpg` command line interface from which you can display the fingerprint using: `fpr`
- This is the data that must be verified with the owner using a secondary communication channel, as both of you must have the same fingerprint associated to the key that is going to be signed. If that happens, you can sign the key using `sign`. In this laboratory **we will suppose that the fingerprint is correct** and sign it.

However, in this phase **you may find that signing a key is impossible for the second user**. *Can you guess why? What is it needed to enable users to sign files using public key cryptography (see theory topics)?* Solve the problem and sign the key. To exit from **GPG** write `quit` or hit CTRL+D. You could be asked if you want to save changes, and you must answer `yes` to finally validate the public key. Once this is finished, you can edit the key again and verify that **GPG** is reporting this key now as `signed: 1`.

WARNING: Keypair generation **REQUIRES** the user invoking the `gpg --gen-key` command to **OWN** the terminal. This can only be achieved by LOGIN to the machine as this user. The easiest way to do it is to use the corresponding `enter` script or to `ssh` from the *Kali* to the *Ubuntu* container. Please note that not even the `su -` command solves this problem! ☹️

Expected results: This activity is finished when you managed to sign an imported key in the second container, solving the potential problems you may find when doing it.

Backing up your private key

To save a raw data or ASCII-armored copy of the private key, you can use the `--export-secret-keys` option: `gpg --export-secret-keys <username> > <private_key_file>`.

To import the private key to your private keyring on another machine (or restore it if the original machine was reinstalled), you can use the `--import` option like this: `gpg --import <private_key_file>`. Note that the exported private key is still protected by the passphrase you used when you created it. To test this, export the private key you created on the *Ubuntu* container and copy it to the `/shared` folder of this container. From the host (your VM), go to the shared folder of this lab and find the exported private key there. **On your real VM** check that you do not have any private key created and import it, checking that now you do have a private key imported in your VM local user keyring. Use the previous lab GPG cheatsheet to remove the imported key from the VM keyring after that. **NOTE**: If you do not like to use your VM, you can do the same operation between the *Ubuntu* and *Kali* containers.

Expected results: This activity is finished when you managed to import an exported private key in another keyring, understanding the purpose of doing this in real life.

Checking a keyring key trust

Even if we are not manipulating a public key ring in this lab, we can use it to check how the keys are downloaded, managed, and verified. To do that, follow this procedure on any of the containers of the lab.

- Go to a public keyring. For example: http://keyserver.ubuntu.com
- Search the key of someone. For example, "Richard Stallman".
- Go to any non-revoked result **User ID (uid)**. Understand what is displayed.
- Search for a result that appears to be not fake. **Click on the public key** and analyze what you see now.
- Click on some of the signatures of the key and understand what you find there.
- To import the public key with `gpg`, use this command: `gpg --keyserver keyserver.ubuntu.com --recv-keys <keyId>` (the one that appear in the webpage after `rsa4096/`).
- After doing that, run `gpg --list-sig <keyID>` and determine if the key you have just imported can be really trusted.
- Use the previous lab GPG cheatsheet to remove the imported key for the local keyring (**WARNING**: multiple public keys could have been imported, check that all are removed! 😊).

Expected results: This activity is finished when you can examine a key from a public keyring and the information the keyring provides, import it, and examine the key you imported.

BLOCK 2. CONFIDENTIALITY AND AUTHENTICATION

Asymmetric encryption for confidentiality

Practical application: *You need to encrypt a file so just one particular person can read its contents*

As explained in the theory topics, to do asymmetric encryption you need to have the public key of the person you want to encrypt data for. Once you have the public key of the receiver in your keyring, you can proceed with the following encryption commands:

- To encrypt files, you can use the `-e` (or `--encrypt`) option along with the `-r` (or `--recipient`) option: `gpg -e -r <username> <filenametocipher>`. For example, if someone wanted to encrypt a file called `file.txt` for the user `redondo`: `gpg -e -r redondo file.txt`
- This will produce an encrypted file called `file.txt.gpg` that only the user `redondo` can decrypt. If you need to change the name of the resulting encrypted file, use the `-o` (or `--output`) option. For example, to call it `file_for_redondo.gpg`, you could use: `gpg -o file_for_redondo.gpg -e -r redondo file.txt`. **NOTE**: Unsigned public keys (see previous section) will receive a warning.
- **TIP**: Remember that writing files in the `/shared` containers directory automatically puts them in a known folder in the host, and vice versa! 😄 .

Decryption of asymmetrically encrypted text

For the recipient to decrypt encrypted data, they need to specify the output file using `-o` and use the `-d` (or `--decrypt`) option. Example: `gpg -o decipheredfile.txt -d cipheredfile.txt.gpg`. The recipient will be prompted to enter the passphrase for their private key. If the correct passphrase is used, the decryption algorithm will proceed, and the original data will be stored in `decipheredfile.txt`.

Expected results: This activity will be finished when you are able to send an encrypted file from the user in the *Kali* machine to the user in the *Ubuntu* machine and correctly decipher it thanks to the previous key importing/exporting operations. You also need to understand what keys you need to do that.

Password-less SSH

Practical application: *You need to stablish an automatic secure connection with a remote machine, so no password is asked when the connection is made although it remains secure*

One of the most common applications of public and private keys is to enable direct login through SSH into a remote machine that has our public key registered and associated to a user identity. This way, every time we try to log into this machine, our public-private keypair will be used to identify us as this user remotely, and no password will be needed. To do this, this activity requires you to use this procedure to generate a pair of `ssh` public/private keys (not the `gpg` ones, a new pair) so whenever the user on the *Kali* container connects to the *Ubuntu* container with its identity the login happens automatically and no password is required (put no passphrase on the generated keypair): https://linuxize.com/post/how-to-set-up-ssh-keys-on-ubuntu-1804/

NOTE: It is not necessary to disable `ssh` password-based authentication, although now you could do it, obtaining some security advantages... 😄

Expected results: This activity will be finished when you are able to connect from the *Kali* container to the *Ubuntu* container using your user identity and no password is required for that.

BLOCK 3. INTEGRITY AND IDENTITY

Digital signatures for integrity

Practical application: *You need to make sure that a file your sent or receive will not be altered or has been altered*

Confidentiality of data is an independent property from integrity. Both can be used together or individually from each other. As said in the theory topics, **integrity is achieved via digital signatures**. There are two types of digital signature that **GPG** can apply.

- A **normal signature**, where the raw binary data of the signature is included with the original data.
- A **clear sign signature** where the signature is added as readable text (a *base64* ascii-armor signature).

Clear signatures

To use GPG for a clear text digital signature use the `--clearsign` option.

- To sign `file.txt` using this method you would use: `gpg --clearsign file.txt`.
- The output is an `.asc` file with the original content plus the digital signature. *(**NOTE**: If you accidentally `cat` a binary file (such a raw `gpg` certificate file), the bash prompt could become corrupted and show unreadable characters. You may fix this typing the command `reset`)*
- On the receiving side, the recipient can verify the signature **provided they have the sender's public key imported** on their keyring. This can be done like this: `gpg --verify file.txt.asc`.

Using a normal binary digital signature

To sign data normally, use the `-s` (or `--sign`) option. Using the same `file.txt` that we used above, we can sign like this with this command: `gpg --sign file.txt`. This will produce the file `file.txt.gpg`, whose contents are composed by our data along with what looks like unreadable random data (**this is the actual raw signature**). Someone with the sender's public key can verify this in the exact same way as a clear sign signature by using the `--verify` option: `gpg --verify file.txt.gpg`.

Expected results: This activity will be finished when you are able to sign data on a container (cleartext and binary) (e. g. *Kali*) and verify it in another container (e. g. *Ubuntu*).

Using a digital signature and asymmetrically encrypted data

Practical application: *You need to send files with secret contents, and you also must be sure that these contents have not been modified*

Finally, a sender may encrypt and sign the data for a recipient, **combining both confidentiality and integrity**

- The sender simply needs to combine the commands we used for asymmetric encryption with the `-s` (or `--sign`) option. The command below signs and encrypts for recipient `redondo`: `gpg -o cryptandsign.enc -s -e -r redondo otherfile.txt`
- Note that again the receiver will need the senders public key on their keyring. The `-d` or `--decrypt` option will automatically do both decryption and signature verification if a file uses both techniques: `gpg -o original_file.txt -d cryptandsign.enc`

The file is now deciphered, and its contents are clear. If the recipient does not have the sender's public key on their keyring for verification, the decryption will still work as usual, but following message will be displayed: *"Can't check signature: public key not found"*.

*(**NOTE**: Beware of the terminal owning warning we previously described, as it may happen here also!* ☹ *)*

Expected Results: This activity will be finished when you are able to send and encrypted and signed file to another user and you can see and verify its contents.

Manually checking the signature of a downloaded program

Practical application: *You need to check the integrity of a file you download to be sure it has no corruption on malicious alteration*

Signatures of files are commonly used to check that programs / documents / etc. have not been tampered with or corrupted when downloaded. Most programs include instructions about how to verify the executable against a signature that is displayed on the website (or downloaded within the executable package) to ensure the executable is not corrupted / tampered. To check how to do that, follow this procedure (you can do this **over the *Ubuntu* VM itself**):

- Go to the download page of a program that offers verification signatures of their downloaded files. For example: http://archive.ubuntu.com/ubuntu/dists/bionic-updates/main/installer-amd64/current/images/netboot/
- Download the `mini.iso` of *Ubuntu* and check its signature with the `sha256sum` tool.
- Go to the page where the actual signatures are listed and open the appropriate file with the signatures in the format you used: http://archive.ubuntu.com/ubuntu/dists/bionic-updates/main/installer-amd64/current/images/
- Compare your result with the one listed there.

Expected Results: This activity will be finished when you are able to ensure that the downloaded program is unaltered according to the author signatures.

Watermarks

Practical application: *You need to mark an image with something that identify it, so it is more difficult that other person claims its authorship*

Although **this is not signing a file**, watermarking images is a procedure to ensure that a copyrighted image is not easily stolen by a third party that claim its authorship. Of course, the image could still be tampered, but removing a watermark is a process that stops some non-tech users from stealing them. It could also be used to create physical watermarked copies of documents so they could not be distributed without it, as removing them from physical copies is way more difficult. We can watermark an image using the *Image Magick* package (`sudo apt install imagemagick` and using its `convert` tool). Two examples that paint rotated red text (although you can easily change the parameters to test other variants) are:

- **Short text big watermark**: `convert -density 150 -fill "rgba(255,0,0,0.25)" -gravity Center -pointsize 80 -draw "rotate -45 text 0,0 <text>" <original image>  <watermarked image>`
- **Two-line watermark**: `convert -density 150 -fill "rgba(255,0,0,0.50)" -pointsize 15 -draw "rotate -15 text 0,200 '<line of text 1>'" -draw "rotate -15 text -25,260 '<line of text 2>'" <original image>  <watermarked image>`

Use these examples to watermark any image you have. **You can do this in the *Ubuntu VM***.

Expected Results: This activity will be finished when you are able to place text to watermark any image you download from the internet

Metadata (anti-signing! 😉)

Practical application: *You need to explore or remove the metadata of any file*

Finally, as we said in the previous laboratory, sometimes metadata can associate a file (like an image) with our user identity without being aware of it (it has our name, our IP, data of one of our devices …). However, there are scenarios (like publishing the document in the web) in which we **COULD NOT** want this association to appear. It means that, even if we have practiced signing documents to associate them with us, sometimes we need to do the opposite and break any association. Metadata is arguably the most forgotten way of having compromising data, so we will end this laboratory teaching you the following operations with the highly popular image metadata manipulation program `exiftool` (`sudo apt install exiftool`), even though **it has NO relation with information signing**. Note that there are **essential file metadata** (the information that **must** be in any file of a type as a requirement) and **extra metadata**, introduced by a program that creates them or another tool. Logically, `exiftool` just removes extra metadata.

- **Consult the current metadata of an image**: `exiftool <image file>`
- **Remove all the metadata of an image**: `exiftool -all= <image file>` (note the space character after the `=`)

Use these examples to consult and remove the metadata of any image you have or download. E. g.: https://biosferadigital.com/sites/default/files/archivos/AK-interior.jpg

NOTE: `exiftool` is **VERY** powerful, and you can check a lot of usage examples here if you are curious:

- https://exiftool.org/examples.html
- https://exiftool.org/exiftool_pod.html

Expected Results: This activity will be finished when you are able to read and remove the non-essential metadata from any image.

BADGES AND SELF-ASSESSMENT

NOTE: You have a version of this badge table in an editable format available in the *Virtual Campus*. You can use this file to easily create a document in the format you want and take extended notes of your activities to create a log of what you did. Remember that this material is key to keep track of your self-evaluation.

Badge Level	Unlocked when	Unlocked?
	Locate the key pair (public and private) generated by GPG managing the user local keyring.	
	View an exported public key file and understand its contents.	
	Import a key from another user and view it in your keyring.	
	You can sign an imported key, understanding what it is used and what you need to sign a key.	
	You can create a clear and a binary signature	
	You can answer this question: *When you export and import a secret key into another keyring. Is the public key also imported/exported?*	
	Know how to remove a public and a private key from a keyring.	
	Know how to check the signature of a downloaded program with `sha256sum`. You also know which algorithms to avoid, to ensure that the signature is not easily falsifiable.	
	You can watermark any image	
	You can read metadata of any image. You clearly understand why this is important for security reasons	
	You can remove the metadata of any image. You clearly understand why this is important for security reasons	
	You can answer this question: *What algorithm was chosen by GPG to generate the key pair? What key size? Do you consider it secure according to the theory of this course?*	
	You can answer this question: *Do you think you can send your public key via email to another person? How? Is that a security problem or not?*	
	You can answer this question: *What can you do when you import a public key from any user?*	
	Understand what is the purpose of exporting a private key and re-importing it in another machine. You can answer this question: *Should you send this key via email? Why?*	
	Be able to log in as a user through SSH using a public key, understanding the process.	
	Be able to sign data using clear and binary signatures and understand the differences.	

	You can answer this question: *When you examine a keyring, what information do you see?*	
	Understand how many keys you need to perform asymmetric encryption for a particular user.	
	You can answer this question: *What would you do if you must remove the possible extra metadata of any image in a web application you have to put into production?*	
	You can send ciphered messages to concrete users that you have imported their public key into your keyring.	
	You can send ciphered and signed messages to concrete users that you have imported their public key into your keyring.	
	You can answer this question: *What should you do with your keypairs if you move to another machine, or reinstall your current one, to avoid data loss?*	
	Once you understand the process of login remotely using your public key, consider that password login through SSH can be totally disabled to any user. If we do that, *is it more secure? Will we prevent certain attacks? What should we do if we want another user to log in now?*	
	You can answer this question: *Is the number of signatures the only reason to trust a certain key in a keyring? What other hints can be found that make the key invalid?*	
	In the process of creating a password-less authentication via SSH, *can't we just copy our generated public key to any remote user and impersonate any of them? If not, in which moment we did something that prevented us to do it?*	
	**You can't touch this**: Your asymmetric cryptography understanding makes you able to use asymmetric encryption to send confidential data privately to any user you want, ensuring also that the data you send has been unaltered. Also, you can establish confidential communication channels with remote machines for selected users. Finally, because of understanding the conceptual process of signing data, you can effectively use watermarks to identify your images if needed and determine if an image may have extra information that can identify something about its author(s).	