Source: Bing Chat AI

Laboratory 4. Applications of Cryptography (II)

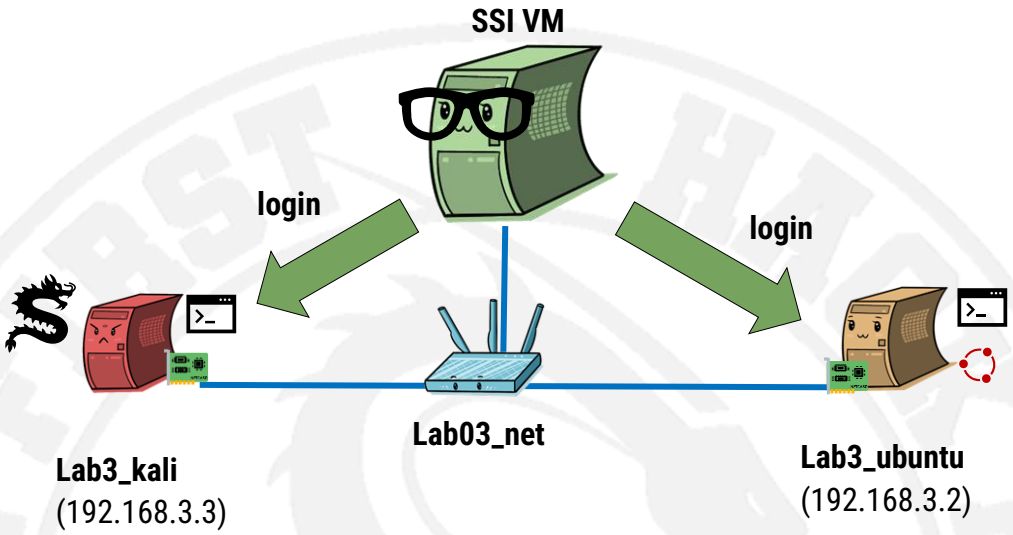
COMPUTER SCIENCE DEPARTMENT. UNIVERSITY OF OVIEDO

Computer Security | 2023 – 2024 (V4.0 *"S-81 Isaac Peral"*)

Content

Infrastructure of this laboratory

This lab uses exactly the same infrastructure as the previous one. This drawing is just here to remind you of what was it like



Block 1: Asymmetric Encryption

Exercise L4B1_KEYPAIR: Generate a key pair (Public – Private)

Description of the activity / Practical application

The idea is **to create the necessary conditions** to allow two users to transfer information to each other in the future using asymmetric encryption.

Expected results

You can **create a separate key pair for each user** who is part of a secret communication that uses asymmetric encryption. You can also answer the questions that are asked during the execution of the exercise.

Additional information required to do it

As we have seen in theory, in order to do a secret communication using asymmetric encryption each participating user must have a generated key pair associated with their identity. The algorithm that we are going to use to create these key pairs is **RSA**, also seen in theory.

RSA works with up to 4096-bit keys. The longer the key, the better, as it improves security against brute force attacks. In addition, the algorithm can encrypt more data with a longer key (although the encryption will be slower).

Using the image below (https://cheatography.com/albertx/cheat-sheets/openssl/), search for the command to generate a 4096-bit RSA key pair.

ASYMMETRIC ENCRYPTION
List elliptic curves available **openssl ecparam -list_curves**
Create 4096 bits RSA public-private key pair **openssl genrsa -out pub_priv.key 4096**
Display detailed private key information **openssl rsa -text -in pub_priv.key -noout**
Encrypt public-private key pair using AES-256 algorithm **openssl rsa -in pub_priv.key -out encrypted.key -aes256**
Remove keys file encryption and save them to another file **openssl rsa -in encrypted.key -out cleartext.key**
Copy the public key of the public-private key pair file to another file **openssl rsa -in pub_priv.key -pubout -out pubkey.key**
Encrypt a file using RSA public key **openssl** pkeyutl **-encrypt -inkey pubkey.key -pubin -in cleartext.file -out ciphertext.file**
Decrypt a file using RSA private key **openssl** pkeyutl **-decrypt -inkey pub_priv.key -in ciphertext.file -out decrypted.file**
Create private key using the P-224 elliptic curve **openssl ecparam -name secp224k1 -genkey -out ecpriv.key**
Encrypt private key using 3DES algorithm **openssl ec -in ecP384priv.key -des3 -out ecP384priv_enc.key**

When you find it, **generate a pair of keys for the user of the** *Ubuntu* container and another for the user of the *Kali container*, calling the generated files differently to avoid future problems. Once you have generated each pair of keys, also look for the option to **extract the public key of each pair from them**, writing it to a separate file that is also called differently for each user. *Do you know which of those files you would have to send to anyone who wants to encrypt information that only you can read? What do you think would be the point of copying the file with the full key pair to another machine you own?*

In the next step of this exercise, check the contents of the two files generated for each user. *Have you seen the size and randomness of the generated keys? Do you think they are keys that are easily crackable by* **john** *as we saw in the previous lab if they were used to encrypt content?*

Finally, just for the purpose of checking what is said in theory, use the commands in the image above to **generate a private key that uses elliptic curve cryptography** from the **secp256r1** curve that is used for the *ECDH (Elliptic Curve Diffie-Helman) algorithm. How big is it? Is it larger or smaller than those used with standard Diffie-Hellman, which are usually between 2048 and 3072 bits?*

Exercise L4B1_ASSYMCRYPT: Using asymmetric encryption to send sensitive messages

Description of the activity / Practical application

It involves **encrypting and decrypting a file** using the concepts of asymmetric cryptography

Expected results

You can **create an encrypted file** so that only a specific person of your choice (from which you have the necessary key to do it) will be able to read it. You can answer these questions:

- *Do you understand that for this encryption to work there has to be a means for everyone to access the real public keys of the interlocutors to whom they want to send the message?*
- *Do you understand, therefore, the essential function of trust rings and certificate authorities when it comes to issuing/validating certificates that associate public keys with people's identities?*
- *Consequently, do you understand why without these entities the scheme would not work?*
- *Also think about the following: what is the most convenient behavior programming-wise, an error you can detect/capture if the decryption is incorrect or that it simply decrypts something, even if the key is invalid?*

Additional information required to do it

To test the encryption, we have seen in theory, look for the appropriate commands in the image above to do these operations:

- Notice that right now we have two **containers with two users,** which we will call U and K, and each of them has its own key pair. Under these conditions we cannot send anything to anyone. To enable communication, think carefully about what we have said in the theory and **give U the correct key of K that allows U to send messages that only K can decipher.** Do the operation to also allow the opposite secret communication, between K and U.
- **Create a very small text file in** the U container.
- **Encrypt the file so that only K can read it.** Check that K can indeed decrypt the file. *What are we assuming for us to claim that only K can do it? What would have to be leaked or stolen for this secret communication to stop working and for a third party to be able to read the contents?*

Finally, try to decrypt one of the files encrypted with the wrong key (for example, try to decrypt it with one of the public keys and then with the private key of the incorrect user). *What happens in each case? Do you know for sure when a decryption operation fails and, therefore, when you have been given the wrong key?*

Block 2: Combining symmetric and asymmetric encryption

Exercise L4B2_KDFPASS: Using a KDF as a password for a symmetric cipher

Description of the activity / Practical application

It consists of **generating a truly strong password for a symmetric encryption**, so that it is no longer crackable as in the previous laboratory.

Expected results

You know how to do strong symmetric encryption **with a very strong key**, just as you would do for real uses.

Additional information required to do it

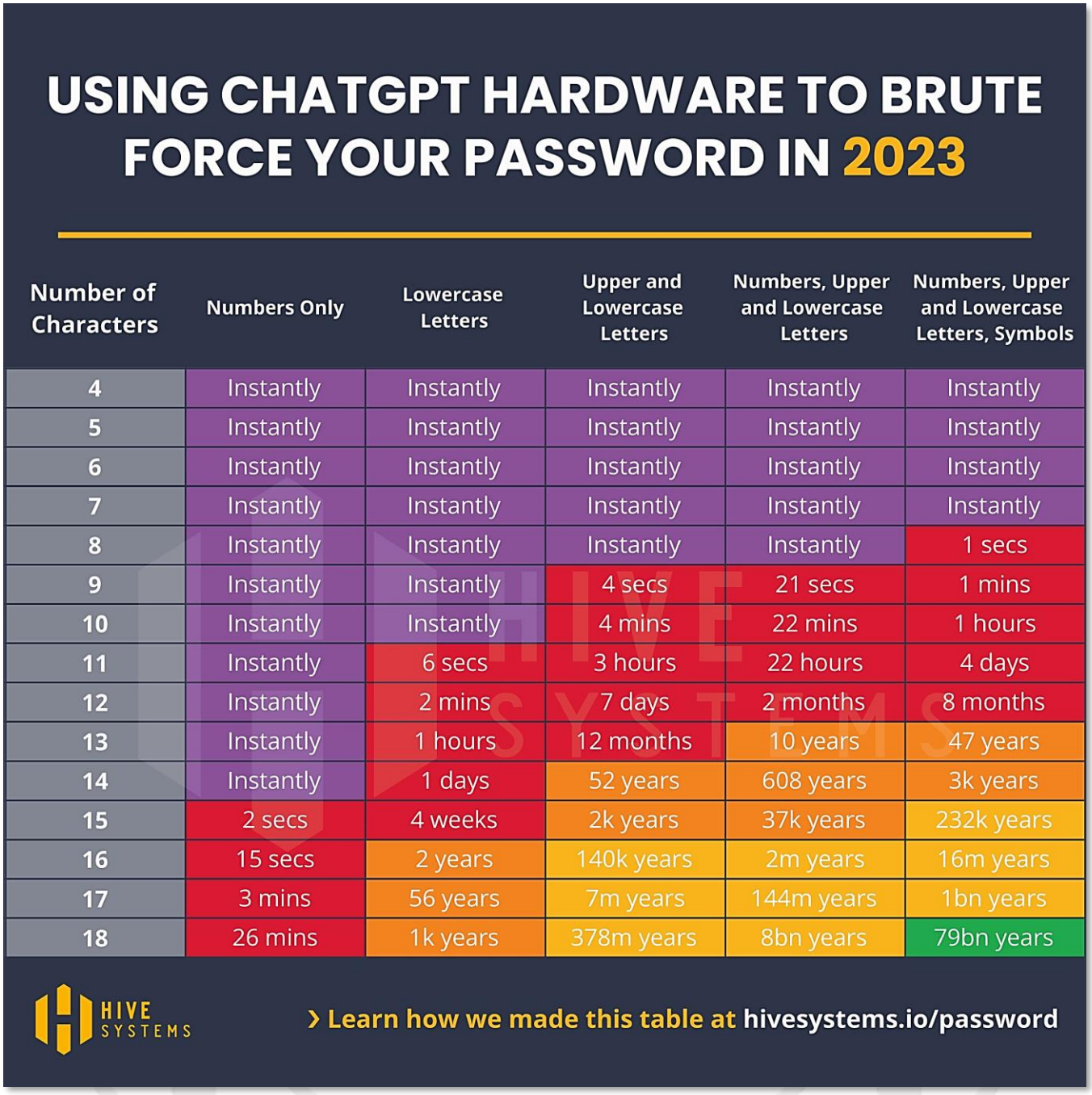
> 🔍 ***An encryption key is a random string of bits explicitly created to encrypt and decrypt data. In this scenario, the more random the string, the better for these purposes. For this reason, having a robust and proven random number generator is essential in cryptography.***

Since manually typing characters complete a 128bit string is not a good way to generate anything random (no, it is not, trust me 😊), we can use one of the concepts seen in theory to generate a truly random string: a ***Key Derivation Function*** (KDF) algorithm. As seen in theory, this algorithm processes the initial key that we give in plain text to create through a series of mathematical operations and rounds a random hash from it. **This random hash is the one we can use as an encryption key**, instead of using "password" as before.

To generate a random key with this procedure we just have to do this: `openssl enc -pbkdf2 -aes-128-ecb -k <key in initial text> -P`. This command asks OpenSSL to create a 128-bit AES key (`-aes-128-ecb`) using a KDF function mentioned in the theory, **PBKDF2** (`-pbkdf2`), and print the result to the terminal (`-P`). To be able to manage it better, we can save it in a file called `csim.priv` or with whatever name you want (please, **save only the private key contents**, not the salt or the `key=` or `salt=` strings).

Now we will be able to encrypt and decrypt the file with symmetric encryption **exactly as we saw in Lab 3,** but this time the key will be passed in a `-K` parameter like `-K $(cat csim.priv)`. Try now to encrypt and decrypt a file you create as a sample (or the same one from the previous lab) with this new strong key to check that everything is working properly.

Finally, use this picture to determine whether or not your new key can be broken using brute-force. Assume that letters can be uppercase or lowercase indistinctly.



Number of Characters	Numbers Only	Lowercase Letters	Upper and Lowercase Letters	Numbers, Upper and Lowercase Letters	Numbers, Upper and Lowercase Letters, Symbols
4	Instantly	Instantly	Instantly	Instantly	Instantly
5	Instantly	Instantly	Instantly	Instantly	Instantly
6	Instantly	Instantly	Instantly	Instantly	Instantly
7	Instantly	Instantly	Instantly	Instantly	Instantly
8	Instantly	Instantly	Instantly	Instantly	1 secs
9	Instantly	Instantly	4 secs	21 secs	1 mins
10	Instantly	Instantly	4 mins	22 mins	1 hours
11	Instantly	6 secs	3 hours	22 hours	4 days
12	Instantly	2 mins	7 days	2 months	8 months
13	Instantly	1 hours	12 months	10 years	47 years
14	Instantly	1 days	52 years	608 years	3k years
15	2 secs	4 weeks	2k years	37k years	232k years
16	15 secs	2 years	140k years	2m years	16m years
17	3 mins	56 years	7m years	144m years	1bn years
18	26 mins	1k years	378m years	8bn years	79bn years

Exercise L4B2_REALASSYMCRIPT: Understanding how asymmetric encryption works in real life

Description of the activity / Practical application

It consists of **replicating the process used by SSL and other similar encrypted information transmission services** over the Internet to combine the use of symmetric and asymmetric encryption to encrypt large amounts of data, by taking advantage of the best of each type of encryption.

Expected results

You can **send a large, encrypted file to someone** and the information needed to decrypt it **by combining symmetric and asymmetric ciphers** as it is done in real life, understanding what the process is like and what is encrypted with what. You can answer these questions:

- *Do you understand that you can now send the encrypted file by one transmission channel, and the key to decrypt it (which will also be encrypted, and only the authorized person will be able to decrypt it) by another, to maximize the security of the transmission?*
- *Do you realize that this hybrid procedure ensures confidentiality and authentication of interlocutors at the same time, so that only the right people can know the secrets?*
- *Do you think that all this is too complex for continuous use, for example in sending encrypted content by email? If you think so, you can use a secure email service such as* ***ProtonMail*** *(https://proton.me/es-es/mail) for your day, which integrates all these concepts, it is free, and gives you all the advantages that are seen in this exercise very easily.*

Additional information required to do it

It happens that, for RSA to work with large data, the **file with the data would have to be split into blocks of 512 bytes** (if the data is smaller than 512 bytes, the block is automatically filled until it reaches this size), otherwise it does not work. Try it by **trying to encrypt a large file** (for example larger than 1 Mb) that you download from the Internet with a public key, as in block 1. *What happens?*

🔍 ***Even if we invest time into splitting a file into 512-byte blocks, as we said in the theory RSA is very slow encrypting large data***

It is time to solve this problem **by combining both cipher types**, to illustrate what we have seen in theory about how they are used in real life. Follow these steps:

- **Encrypt the entire file with symmetric encryption** and a really strong key as we saw in the previous exercise. If instead of the AES ECB mode you decide to use the CBC mode as seen in **Lab 3,** remember to generate an IV with `openssl rand` as we saw then, to pass it in an additional -iv parameter.
- **Encrypt the key you used to make this symmetric encryption** (e.g., `csim.priv` from the previous exercise) with the appropriate key of the recipient key pair so that only the recipient can see this decryption key (you know which one 😉).
- **Send the recipient** both the file with the data, encrypted with symmetric cryptography, and the file with the key you need to decrypt it, that you have just asymmetrically encrypted.
- Do what is necessary to obtain the decryption key of the file and, with it, the original file. *Do you now understand how these exchanges that use both methods work?*

NOTE: You can actually use this exercise as proof that you have understood how these types of ciphers are used, because it forces you to work with both .

Block 3: Integrity and digital signatures

Exercise L4B3_AE: Using *Authenticated Encryption (AE)*

Description of the activity / Practical application

It consists of understanding **how to use authenticated encryption** and its special features to better understand the theory concepts.

Expected results

You can use authenticated encryption to encrypt content and **understand the benefits it brings** by using *CyberChef*. You can answer these questions and the ones posed in the exercise text:

- *What happens if I change a character in the encryption key or tag?*
- *What happens if I change a character in the encrypted input when I try to decrypt it?*

Additional information required to do it

To practice with *Authenticated Encryption*, one of the things we need to do is **select a symmetric encryption algorithm with this feature.** One of them is **AES-256-GCM** (*Galois Counter Message*, one of the three ways to create a MAC shown in the theory). Try to encrypt any file with this algorithm with `openssl` with this command: `openssl enc -aes-256-gcm -nosalt -p -in file.txt -out file.out` *Does it work?*

The thing is, `openssl` has decided not to implement AE or AEAD encryptions for the time being for a number of reasons. The main reason is that the output cannot be fully validated until all the input is available, which precludes its use with streaming traffic, as is typically the case when using SSL.

In other words, it is only possible to know if the traffic is correct when you have all the traffic that is going to be emitted in full. If the traffic is arriving gradually, the receiver processes what is arriving (and acts accordingly), and in the end of the transmission it is discovered that it was corrupted, all the work that the receiver has done with the data that was arriving will have to be undone, which is not only problematic to implement, but can also be a very serious security problem in this scenario.

But that does not mean we cannot try it! ***CyberChef***, which we already learned **to use in Lab 3**, has a way to encrypt with **AES-GCM** that we can use, since the input it supports is **finite and completely known** in advance (either we write it ourselves or we load a file with it). To try it out, open the tool and follow these instructions:

- The algorithm **needs a key**. It uses a strong key **created with a KDF** as we saw in the previous section. When you generate one, **save it for later**.
- You also **need an IV.** You can generate a random one like we saw in **Lab 3** when we talked about how to use the AES CBC mode. If you generate a 16-bit one, the tool will automatically select **AES-128-GCM** for you, which is perfectly adequate for this exercise. When you generate one, **save it for later**.

- You do not need to add ***Additional Authenticated Data*** (for this exercise it does not matter if you use AE or AEAD).
- The ***Input*** is recommended to be entered ***Raw*** and can be a simple text string. The ***Output*** can be selected in ***hexadecimal format***.
- **Encrypt the content** and copy the result, both the output and the *Tag*.
- Now, using the key and the IV that you have used for encryption, **decrypt with *CyberChef*** what you obtained as ciphered output (not the tag!), to check that it works. Do not forget that the output also gave you a string of characters called a T*ag*, which you must also copy and include as a parameter when decrypting, as it is **what makes it possible to authenticate the encrypted data**.
- **Alter any of the data** (the input, the key, the IV, the Tag...). *Is the encryption issue detected?*
- Now encrypt *any data* **in AES-CBC mode** with CyberChef **(**unauthenticated encryption, the one we saw in **Lab 3**) using the key and IV you want (you can reuse the ones you already have) and write down the result. Try to decrypt it later with that same data and see if it works. Now, **replace some characters from the IV with** others and see what happens. *Is the problem detected? Do you understand, then, why authenticated encryption is necessary and the difference between authenticated and unauthenticated encryption?*

BACK
1
2
3
4

Exercise L4B3_DSIG: Digital signatures for integrity

Description of the activity / Practical application

It consists of understanding the usefulness of file signatures, as well as the procedure to generate and validate them correctly.

Expected results

You can generate the digital signature of a file and verify it correctly. You can answer these questions:

- *Do you think it makes sense to send the file and the signature separately to a recipient in any specific case?*
- *What happens if I change a character in the correctly received file after validating it, and try to validate it again?*

Additional information required to do it

The objective of this exercise is to **practice with the use of digital signatures** to ensure the authenticity of a message, in the sense that it has not been modified since it was issued.

To do this, it is worth remembering that a digital signature of a message is nothing more than encrypting the hash of the message with the sender's private key, so that, when it reaches its destination, the sender's public key can be used to decrypt the received hash, and compare it with that of the message that has been received calculated at the destination.

Any discrepancies in the signature check indicate that the message has been altered or corrupted in some way. `OpenSSL` allows you to do the signature and verification processes described very easily.

DIGITAL SIGNATURES
Generate DSA parameters for the private key. 2048 bits length **openssl dsaparam -out dsaparam.pem 2048**
Generate DSA public-private key for signing documents and protect it using AES128 algorithm **openssl gendsa -out dsaprivatekey.pem -aes-128-cbc dsaparam.pem**
Copy the public key of the DSA public-private key file to another file **openssl dsa -in dsaprivatekey.pem -pubout -out dsapublickey.pem**
To print out the contents of a DSA key pair file **openssl dsa -in dsaprivatekey.pem -text -noout**
Signing the sha-256 hash of a file using RSA private key **openssl dgst -sha256 -sign rsakey.key -out signature.data document.pdf**
Verify a SHA-256 file signature using a public key **openssl dgst -sha256 -verify publickey.pem -signature signature.data original.file**
Signing the sha3-512 hash of a file using DSA private key **openssl pkeyutl -sign -pkeyopt digest:sha3-512 -in document.docx -inkey dsaprivatekey.pem -out signature.data**
Verify DSA signature **openssl pkeyutl -verify -sigfile dsasignature.data -inkey dsakey.pem -in document.docx**
Create a private key using P-384 Elliptic Curve **openssl ecparam -name secp384r1 -genkey -out ecP384priv.key**
Encrypt private key using 3DES algorithm **openssl ec -in ecP384priv.key -des3 -out ecP384priv_enc.key**
Sign a PDF file using Elliptic Curves with the generated key **openssl pkeyutl -sign -inkey ecP384priv_enc.key -pkeyopt digest:sha3-512 -in document.pdf -out signature.data**
Verify the file's signature. If it's ok you must receive "Signature Verified Successfully" **openssl pkeyutl -verify -in document.pdf -sigfile signature.data -inkey ecP384priv_enc.key**

Therefore, to practice the use of signatures we need a pair of public and private keys of an issuer (we use the ones we generated in block 1) and use one of the commands in the previous image (https://cheatography.com/albertx/cheat-sheets/openssl/) to sign any file and write the signature to a separate file. Please check the theory topics to know which algorithm can be consider secure to generate signatures.

- Once this is done, send to the receiver of the message **both the message itself and its signature**.
- The receiver should now **use the sender's public key** (which should also be in their possession, and if not, you can copy it) to verify that the message they received has not been altered since it was issued.
- Finally, check what happens if you now modify the received message in any way and try to validate it again.

This is where the main problem with this encryption scheme comes in. When someone sends you a signed message and a public key, *why should you trust this public key? What if the public key and signed message **were altered or replaced during transmission**?* As we saw in the theory, this is solved with **certificates and a public key infrastructure** (PKI) (or a ring of trust) that validates the key of a user X. If we do not have them, we depend on the good faith of the issuers to trust that they are who they say they are. And experience says that good faith and security are not good friends 🙁. The following exercise will try to prove it to you.

Exercise L4B3_CERTSIGNPDF: Sign a PDF with an auto-generated certificate

Description of the activity / Practical application

It consists of understanding the process of **generating a certificate that represents someone** and how it can be used to **digitally sign content** with it. However, in the process, you should also understand that, without a PKI, this signature has very little or no value.

Expected results

You can **understand the complete certificate generation process** and answer the following questions, in addition to those proposed in the exercise:

- *Do you have a good understanding of the differences between a CRT certificate and a PFX certificate, and what each one contains?*
- *Do you understand why you cannot sign with a CRT certificate?*
- *Are you now aware of the problem of not having a certificate authority when trying to sign content with a certificate?*
- *Did you know that you already have the means to sign documents legally and validly thanks to your DNIe?*

Additional information required to do it

With `openssl` you can directly generate **X.509 type** digital certificates like the ones seen in theory, which will have a `.crt` extension.

A certificate with this extension contains the public key of a person or entity, whose data (name, company, email, domain, etc.) are specified in the different fields of the structure of the X.509 format that form the file contents. `.crt` is used, for example, to establish secure connections from the web server to a browser when using the HTTPS (HyperText Transfer Protocol Secure) protocol, since only a public key is required for that.

The first part of this exercise consists of using the content of this image (https://cheatography.com/albertx/cheat-sheets/openssl/) **to generate a CRT certificate** in the name of whoever you want, inventing the data (it can be a celebrity, a classmate, a teacher, a B account of a *streamer*... It doesn't matter).

DIGITAL CERTIFICATES
Generating a CSR file and a 4096 bits RSA key pair **openssl req -newkey rsa:4096 -keyout private.key -out request.csr**
Display Certificate Signing Request (CSR) content **openssl req -text -noout -in request.csr**
Display the public key contained in the CSR file **openssl req -pubkey -noout -in request.csr**
Creating a Certificate Signing Request (CSR) using an existing private key. *This can be useful when you need to renew the public digital certificate without changing the private key.* **openssl req -new -key private.key -out request.csr**
Create EC P384 curve parameters file to generate a CSR using Elliptic Curves in the next step. **openssl genpkey -genparam -algorithm EC -out EC_params.pem -pkeyopt ec_paramgen_curve:secp384r1 -pkeyopt ec_param_enc:named_curve**
Create a CSR file using Elliptic Curve P384 parameters file created in the previous step. *Instead of using RSA keys.* **openssl req -newkey ec:EC_params.pem -keyout EC_P384_priv.key -out EC_request.csr**
Create a self-signed certificate, a new 2048 bits RSA key pair with one year of validity **openssl req -newkey rsa:2048 -nodes -keyout priv.key -x509 -days 365 -out cert.crt**
Create and sign a new certificate using the CSR file and the private key for signing (you must have a openssl.cnf file prepared) **openssl ca -in request.csr -out certificate.crt -config ./CA/config/openssl.cnf**
Display PEM format certificate information **openssl x509 -text -noout -in cert.crt**
Display certificate information in Abstract Sintax Notation One (ASN.1) **openssl asn1parse -in cert.crt**
Extract the certificate's public key **openssl x509 -pubkey -noout -in cert.crt**
Extract the public key's modulus in the certificate **openssl x509 -modulus -noout -in cert.crt**
Extract the domain certificate from an HTTPS/TLS connection **openssl s_client -connect domain.com:443 | openssl x509 -out certificate.crt**
Convert a certificate from PEM to DER format **openssl x509 -inform PEM -outform DER -in cert.crt -out cert.der**
Checking whether the certificate pubic key matches a private key and request file. One step per file. Must match in the output hashes. **openssl x509 -modulus -in certificate.crt -noout | openssl dgst -sha256** **openssl rsa -modulus -in private.key -noout | openssl dgst -sha256** **openssl req -modulus -in request.csr -noout | openssl dgst -sha256**

Once you have generated the `.crt` certificate, you can export it to your host and **view its properties**, for example on *Windows*. *What does the OS say about the certificate? Is it trustworthy?*

🔍 ***However, we are forgetting an important detail, and that is that we want a certificate to sign documents and we can't do it with a CRT one. The reason is simple: as we said before, this type of certificate does not have the private key of its owner, so it is not useful for signing. Do you remember why?***

To sign we need **another type of certificate**, also of type X.509: the ones with a `.pfx` extension. A ***PFX*** (*Personal Intelligence eXchange*) **certificate is a password-protected certificate**. And the reason is simple to understand; it **contains both the public and the private key** of the person it represents. It also contains

other things, such as the certificate authority certificate that validates it, but that does not matter now. What is important is that you understand that a `.pfx` is something like a backup of a person's certificate with its private key, and that it is what we need to sign documents.

How to get a `.pfx` *from a* `.crt`*?* If you notice, when you generated the `.crt` certificate (if you did it correctly 😊) the generated private key **was saved in a separate file from the CRT certificate itself** (which only has the corresponding public one, remember), so you can save both separately with different permissions. *Do you remember why this is necessary?*

You can use the `.crt` and the file with the private key to join them together by forming a `.pfx` with this command: `openssl pkcs12 -export -out <name for the PFX certificate>.pfx -inkey <the private key file you generated when creating the CRT> -in <CRT certificate generated earlier>`

When you create the PFX certificate **, you will be asked for a key to protect it**. You can give any password; in this test it only matters that you remember it (in real life it better be a strong one!)

"Fake" signatures: Signing a PDF with a Self-Signed Certificate

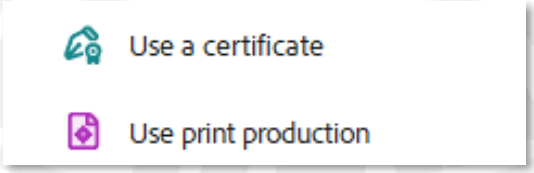
You can imagine that this PFX certificate that you have just generated **has no validity** since no CA endorses it (you just created it yourself!), so in principle it could not be used to sign on its own.

🔍 ***We're going to sign PDFs with Acrobat Reader which, although it's the most popular PDF reader, isn't available for Linux, only for Windows and MAC. We're going to do the example with Windows.***

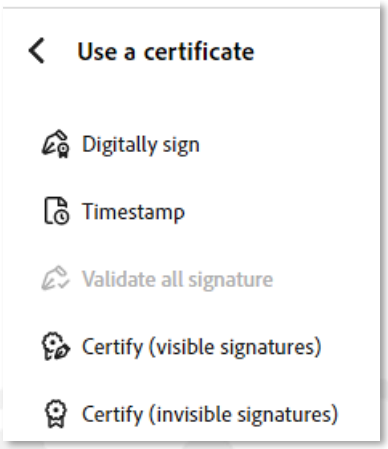
But, even with no known CA involved in its creation, ***Windows* validates any certificate you import into its local certificate store**. In other words, any certificate you import into a *Windows* will be perfectly valid for that *Windows*, even if there is not a CA to sign it. *Do you understand why that can be a security issue?*

Therefore, for this exercise **double click on the certificate**, follow the import tutorial (remember to enter the key you put when creating the PFX), **leave all the options by default**, and check that everything is correct. Now you can open a PDF (this exercise bulletin, for example 😊) to sign it with.

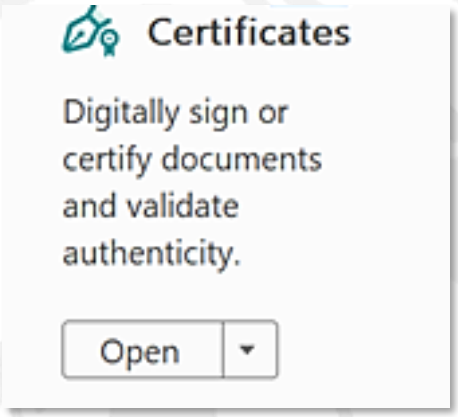
Adobe Reader does not have the tool to sign with certificates very accessible. Depending on the version you have, it changes where it is. For the most modern one, you will have a toolbar on the left where you have to go to the end and click on "***View more***" until this option appears.



Within which you will be able to digitally sign with the certificate:



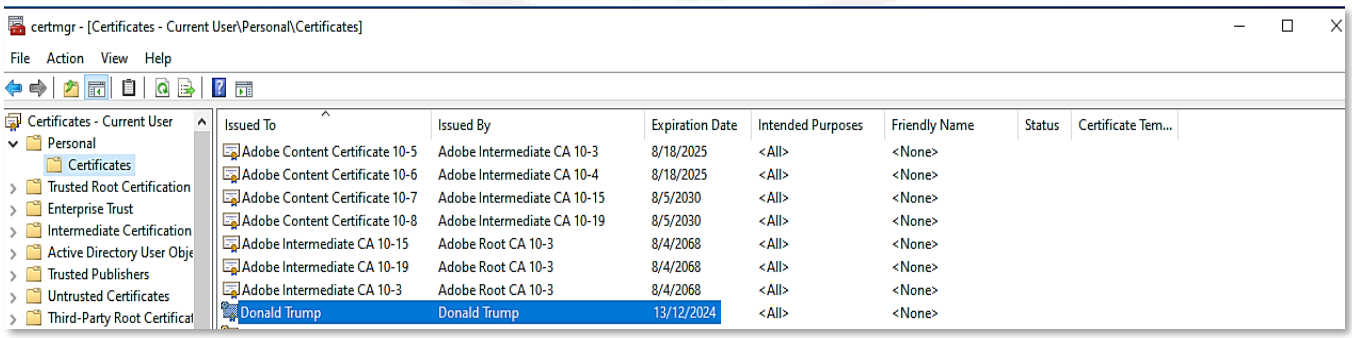
In the older version you have to go to the *Tools* menu that appears to the left of the tabs of all open PDFs and locate this option:



🔍 ***In either case, you have to paint a rectangle with your mouse on the part of the document where you want the signature.***

If you have done it correctly, the certificate that we have created in the name of a random person will appear as an option to sign. Optionally, you can **lock the file**, preventing anyone from modifying it once you sign it. It is very useful in real life, but it is not relevant for this exercise. *Is this signature valid? Do you understand why?*

Now delete the imported certificate by opening ***the "User Certificate Manager***" (“***Administrador de Certificados de Usuario***”) program on your *Windows* (search for it in the start menu by name, it is easier). The certificate (in my case generated in the name of Donald Trump 😊) is in this folder if you have imported it with the default options

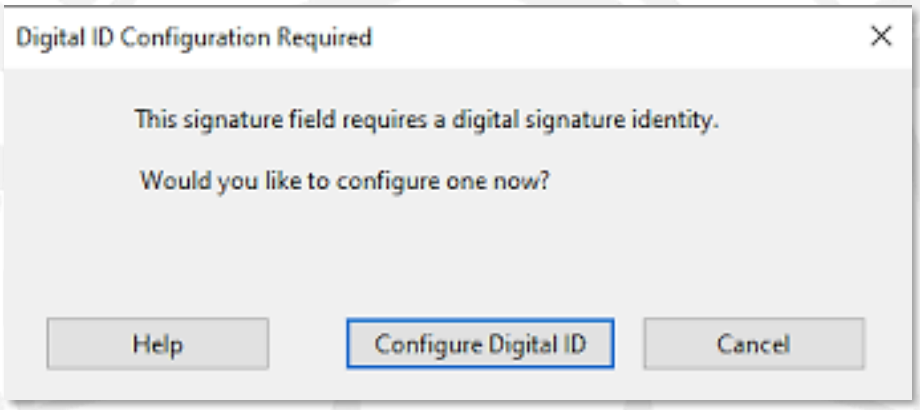


Open the PDF you just signed again. *Is the signature valid now?* To make sure you have a good understanding of what is going on, *What do you think would happen if you now move the document to a different machine? Would the signature be valid?* What would you have to do to make it so? *Do you now understand the important work done by the CAs that are accessible via the Internet from anywhere or those that, having been named trusted by the manufacturers of the OS, install their certificates in each of their OS installs around the world to validate the certificates they issue?*

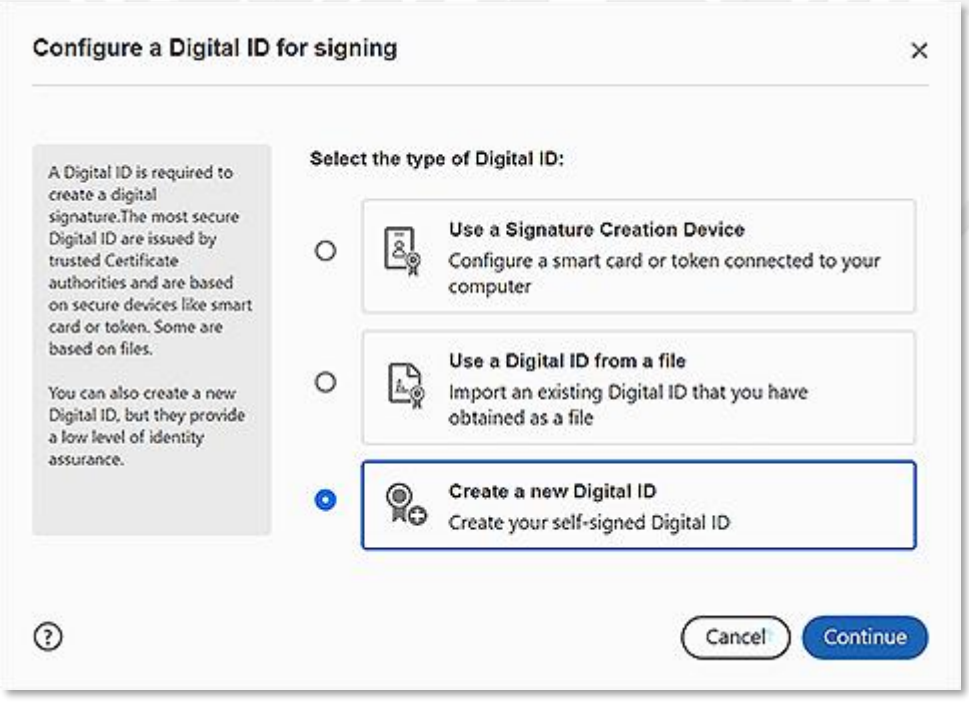
Signing "for real": Free and valid certificates you can get for your needs

🔍 ***This section does not require you to really do any work now, it is for using these concepts in real life.***

Can you sign documents or, in general, have valid certificates at no cost? Yes. And here we are going to explain how. When you try to sign a document in *Adobe Reader* and **you do not have certificates installed before**, you will see this screen:



And then you will have three signature options:



- **First option**: The easiest way to use it to sign is with your electronic ID (Spanish DNIe). You need to have the PIN that they gave you at the police station when you got it, that the certificate is not expired (it expires every 2 years, regardless of when the ID expires, in which case you will have to go to a renewal machine at the police station to renew it) and buy an ID reader. With this you will be able to sign what you want legally, because **your identity is validated by the state.**
- **Second option**: Allows you to use a valid certificate that you have requested, for example, from the **FMNT:** https://www.sede.fnmt.gob.es/certificados/persona-fisica, the entity that will be in charge of validating it. You have to go through the same process that we saw with the self-signed certificate in order to use it
- **Third option**: The same thing we have done with `openssl` ourselves (self-signed certificate) but created by *Adobe*'s own *software* and with the same restrictions and problems, so it is not useful for signing documents legally.

🔍 ***Do you want to support HTTPS on a web server you manage at no cost? Check out Let's Encrypt (https://letsencrypt.org/es/) and, if you have a valid domain online, Certbot (https://certbot.eff.org/) to automate its installation and management. But this is a topic more linked to the subject Server and Network Administration. Or maybe it will be useful in your TFG if you rent a free or very cheap domain...***

Block 4: The concepts of Confidentiality, Integrity and Authentication (CIA) in your day-to-day life

Exercise L4B4_CERTSSH: SSH without a password

Description of the activity / Practical application

You need **to establish automatic secure connections** with a remote machine, so that no *password is required* when the connection is made, and the connection remains secure

Expected results

You can connect from the *Kali* to the *Ubuntu* container using your current user identity and **not be asked for the password to do so.** You can answer these questions:

- *Why do you think that once you transferred the public key to the target system, a password is no longer required, and the traffic is still encrypted?*
- *What is being used to ensure authentication and confidentiality?*
- *Do you understand why users on the source and destination system do not necessarily have to be named the same for it to work?*

Additional information required to do it

One of the most common applications of public and private keys is **to enable direct login (no password is requested) via SSH on a remote machine that has our public key registered and associated with a user identity**.

🔍 ***This way, every time we try to log in to this machine, our public-private key pair will be used to identify us as this user remotely, and we won't be asked for a password***

To do this, you need to follow this procedure: https://linuxize.com/post/how-to-set-up-ssh-keys-on-ubuntu-1804/. There `ssh-keygen` is used to generate a pair of RSA public/private keys for the user who wants to *login*, which we will call **USERKALI**, although the RSAs that we have generated in a previous exercise can be used. When generating the keys, do not put any password to read on the generated key pair when the process prompts you to do so.

Once these keys have been created**, we will transfer the generated public key to the target system,** validating ourselves with a valid user identity in that system, which we will call **USERUBUNTU.** If we do so, every time **USERKALI** connects to the *Ubuntu* container via SSH specifying that it wants to connect as **USERUBUNTU**, the login occurs automatically, and the user's password will not be requested.

🔍 ***NOTE: You don't need to disable password-based authentication in `ssh` (both forms can "coexist") although you could do so now, gaining some security advantages...*** 😉

🛠 Exercise L4B4_CHECKSIGN: Manually checking the signature of a downloaded program

🥷 Description of the activity / Practical application

You need **to check the integrity** of a file you have downloaded, to make sure it **has not been corrupted** or maliciously altered by someone

🎖 Expected results

This activity will end when you can make sure that the downloaded program **has not been modified**, using the signature provided by its author.

📖 Additional information required to do it

File signatures are commonly used to verify that programs/documents/etc. have not been tampered with or corrupted when downloaded.

🔍 ***Many programs include instructions on how to check their executable against a signature displayed on the website (or downloaded into a compressed file along with the executable) to ensure that the executable is not corrupted/tampered with.***

To find out how to do this, follow this procedure (you can do it **in the Ubuntu *virtual machine itself):***

- Go to the download page of a program **that offers verification signatures for their downloadable files**. For example: http://archive.ubuntu.com/ubuntu/dists/bionic-updates/main/installer-amd64/current/images/netboot/
- Download the Ubuntu `mini.iso` (or the file you have found that offers to validate its signature) and verify its signature with the `sha256sum` tool.
- Go to the page where the program 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 to the one that appears there.

🛠 Exercise L4B4_WATERMARK: Watermarks

🥷 Description of the activity / Practical application

You need to mark an image with something that identifies it, so that **it is harder for someone to lie and say it is theirs**

Expected results

This activity will end when you can **watermark any image** you download from the internet. You can answer these questions:

- *Where would you put the watermark to make your work to be misappropriated as difficult as possible?*
- *Would you use steganography combined with this for the same purpose?*

Additional information required to do it

Although **this is not signing a file,** image watermarking is a procedure so that a copyrighted image cannot be so easily stolen by a third party that claims its authorship.

Of course, the image could still be manipulated, but removing a watermark is a process that would prevent some non-tech-savvy users from stealing them.

A watermark could also be used to create physical copies of documents that are watermarked and cannot be distributed without it, as removing them from physical copies is much more difficult. We can mark up 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 try other variants) are:

- **Short text, large 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 virtual machine.***

Exercise L4B4_META: Metadata

Description of the activity / Practical application

You need **to investigate or remove the metadata** from any file

Expected results

This activity will end when you can **read and remove the non-essential metadata** from any image (or the file type you want). You can answer these questions:

- *Do you realize that there will be cases where you want to perfectly identify the image with your name, etc. in the metadata, and cases where you want the opposite? Can you think of examples of both?*
- *How useful do you think it would be to introduce encrypted data with a strong algorithm as metadata in a file, as we saw before?*
- *Have you ever thought about what applications could have knowingly entering fake metadata into a file?*

Additional information required to do it

Sometimes metadata can **associate** a file (such as an image) with our user identity (it can contain our name, our IP, data from one of our devices...). There are times when this can be desirable and times when it can be **very counterproductive**.

🔍 ***Sometimes you'll need to indicate that a file is yours and other times you'll need to break any association with your identity... It all depends on the purpose, where it is published, the context...***

Metadata is arguably the most forgotten way to **have compromising data without knowing it**, so we will finish this lab by showing you the following operations with the popular image metadata manipulation program `exiftool` (`sudo apt install exiftool`). Note that there is **essential file metadata (**information that **must be in any file of a type** as a requirement**)** and **additional metadata**, **introduced by a program** that creates it or another tool. Logically, `exiftool` only removes additional metadata.

- **Query the current metadata of an image**: `exiftool <image file>`
- **Delete all metadata from an image**: `exiftool -all= <image file>` (be careful because there is a space after the `=`)

Use these examples to query and remove metadata from any image you own or download. For example: https://biosferadigital.com/sites/default/files/archivos/AK-interior.jpg**.** Then, use these examples to enter a value for any metadata you want (even if you make it up) into the image:

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

NOTE: The metadata **is NOT related to the cryptographic signature of information concept we saw earlier.**

✍ Badges & Self-Assessment

NOTE: You have a version of this badge table in editable format available on the *Virtual Campus*. You can use this file to create a document in any format you want and take extended notes of your activities to create a log of what you have done. Remember that the material you make can be taken for laboratory tests.

Badge Level	Unlocked when	Unlocked?
	You can create a pair of keys for a specific user and understand the need for protection that each one has	
	You can view a generated key file and understand its contents.	
	Learn how to generate a symmetric encryption key from a KDF algorithm	
	Understand how authenticated encryption works and its particularities	
	You can create a digital signature for any document and use it to check that it has not been altered	
	You learn how to check the signature of a downloaded program with `sha256sum`. You also know which algorithms to avoid making sure the signature is not easily forged.	
	You can watermark any image and use it to try to protect your intellectual property	
	You can read the metadata of any image. You have a clear understanding of why this is important for security	
	You can remove metadata from any image, and you clearly understand why this is important for security reasons	
	You are able to encrypt and decrypt information with asymmetric encryption	
	You can answer this question: *Do you think you can email your public key to someone else? And publish it on the Internet? Is it a security issue or not?*	
	You understand the process of generating a certificate and using it to sign a PDF	
	You understand the purpose of copying a private key and reusing it on another machine. You can answer this question: *Should you send this key via email? Why?*	
	You are able to log in as a user via SSH using a public key, understanding the process.	
	You understand that a self-signed certificate, even if you sign a document, does not really have legal validity and why	

	You know where to get a valid, legal, and free certificate to sign what you need in real life	
	You understand how many keys you need to perform an asymmetric encryption process for a given user.	
	You can answer this question: *What would you do if you had to remove any additional metadata from any image in a web application that you had to put into production?*	
	You can send encrypted messages to specific users whose public key you know and receive a response that only you can decrypt	
	You can send encrypted and signed messages (both) to specific users whose public key you know.	
	You can answer this question: *What should you do with your key pairs if you go to another machine, or reinstall your current one, to avoid data loss?*	
	You can answer this question: Once you understand the process of logging in remotely using your public key, consider that password login via SSH can be completely disabled for any user. If we do that, *is it safer? Will we prevent certain attacks? What should we do if we want another user to log in under these circumstances?*	
	You understand the limitations of asymmetric encryption and know how to use a combination of symmetric and asymmetric encryption to solve them	
	You can you answer this question: In the process of creating passwordless authentication via SSH, *can't we just copy our generated public key to any remote user and impersonate any of them?* If not, *at what point did we do something that prevented us from doing so?*	
	**You cannot touch this**: Your mastery of asymmetric cryptography allows you to use asymmetric encryption to send sensitive data privately to any user you want, while also ensuring that the data you sent has been unaltered. In addition, you can establish confidential communication channels with remote machines for specific users. Finally, because of the general understanding of the process of ensuring CIA information, you can effectively use watermarks to identify your images if necessary and determine if an image may have additional information that can identify something about its authors.	