**Networks and Internet Applications**

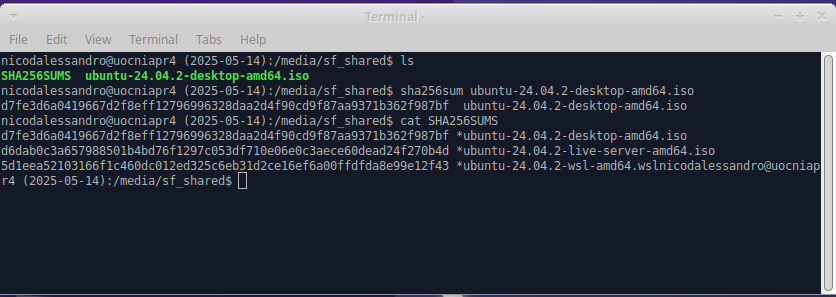
Practical Exercise 4 – PR4

Nicolas D’Alessandro Calderon

**First Part (maximum qualification C-): integrity**

**Exercise 1**

1. We can observe that the hash calculated with sha256 sum matches the one published in SHA256SUMS for this ubuntu image downloaded.



1. The SHA family or Secure Hash Algorithm includes several different versions:

* **SHA-0**: Is the first version, but it is currently not used because it has security problems.
* **SHA-1**: Is more secure than the previous release, but it is also currently broken and not recommended to use.
* **SHA-2**: That includes *SHA-224, SHA-256, SHA-384* and *SHA-512* are stronger and still used today.
* **SHA-3**: Is the newest version designed as an alternative to *SHA-2*.

Each version creates a checksum (hash) with a specific length:

|  |  |
| --- | --- |
| **SHA Version** | **Hash Length** |
| SHA-0 | 160 bits |
| SHA-1 | 160 bits |
| SHA-224 | 224 bits |
| SHA-384 | 384 bits |
| SHA-512 | 512 bits |
| SHA-3 (256) | 256 bits |

* **SHA-0** has a design problem, and it has been proved with several vulnerabilities. It is not safe because the attackers cand find what is called collision (two different files with the same hash). It has been deprecated.
* **SHA-1** was safer than **SHA-0,** but it is also broken. In 2017 Google showed a collision attack on SHA-1, so it has been proved to have the same vulnerabilities as the previous version. Today, SHA-1 is not secure and shouldn´t be used.

1. A computer screen shot of a computer program

   AI-generated content may be incorrect.
2. No, it is not possible to obtain the original text from the hash, because a hash function like SHA-256 is **one-way**. This means that we can create a hash from a text, but we can´t go back from the hash to the original text.

Hash function does not save information about the original file, this means that hashes are good to check integrity but not for reading or recovering the original data.

**Second Part (maximum qualification C+): symmetric encryption**

**Exercise 2**

1. A strong symmetric key should be hard for the attackers to guess or break. Two features that can make the key difficult to break are:
2. **Long key**: If the key is longer, it will be harder to guess or break. Dor instance, a 128-bit key has more possible combination that a 64-bit key making it stronger against brute force attacks (when someone tries many keys until one works).
3. **Random key**: A good key is random, meaning that it should not be based on patterns, names or something easy to guess. For example, a key generated with random characters such as **B9!fZ@73\*Lx#** is going to be much harder to guess that soothing like **nicolas123**.
4. Three commonly used symmetric algorithms for encryption are:
5. **AES Advanced Encryption Standard - Key length 128, 192 or 256 bits.**

This is probably the most used symmetric algorithm today. It is known for being fast and secure. It is used in HTTPS, VPNS and Wi-Fi encryption.

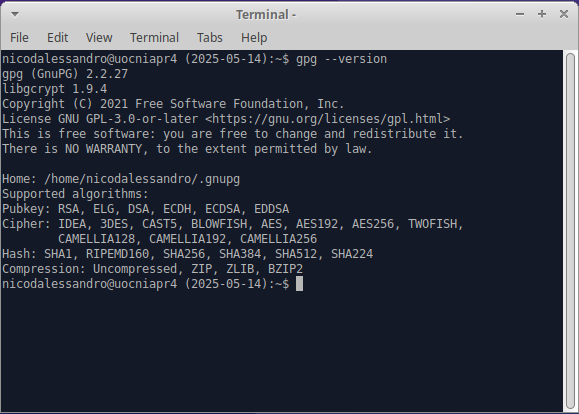
1. **Camellia - Key length 128, 192 or 256 bits**

This is similar to AES in terms of security and speed, but it is used in some systems where AES is not supported such as license or law restrictions.

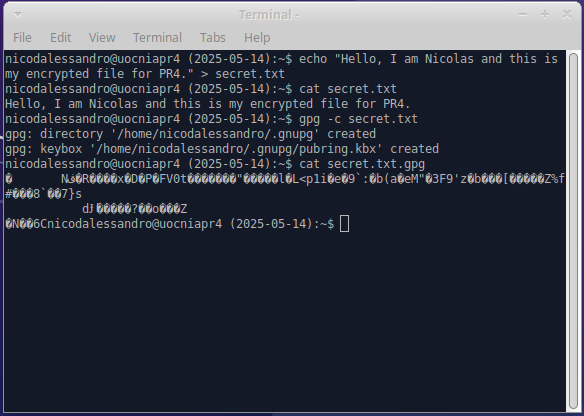
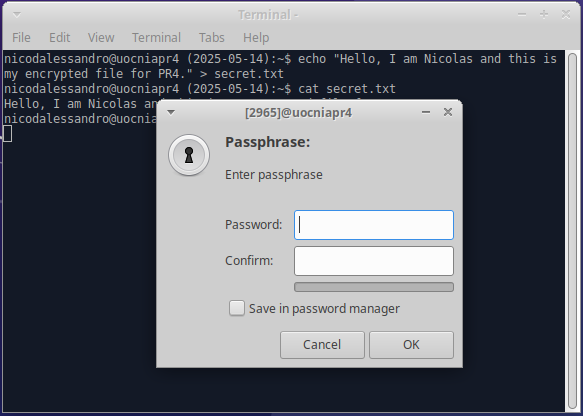
1. **Blowfish - Key length up to 448 bits.**

This algorithm was very popular in the 90´s. Even if it´s still safe to use, is slower than AES so it is less used today.

1. We can observe in the image the supported algorithms:



1. First, I created a **secret.txt** text file with the echo command. Then I encrypted the created file using GPG and symmetric encryption. GPG asked for a password and created the **secret.txt.gpg** file. When I try to open the **.gpg** file with **cat** command I see unreadable characters, showing that the file is encrypted and can´t be read without the password:



1. We can use the command **gpg -d [our-encrypted-file.gpg].** GPG asks for the password, and after entering it correctly, we will see the original message:

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

1. As we saw in the previous exercise, the default algorithm used when none a specific parameter added is AES256 (AES algorithm with a 256-bit key):

**A screenshot of a computer

AI-generated content may be incorrect.**

1. To encrypt a text file using the CAMELIA256 algorithm, I first created the text file and then we add the parameters **gpg -c --cipher-algo CAMELLIA256 camelliaPR4.txt**

**gpg** is the encryption program.

**-c** is to indicate symmetric encryption with a password.

**--cypher-algo CAMELIA256** tells to gpg to use the CAMELLIA256 algorithm instead of the default AES256.

**[filename-to-be-encrypted.txt]** is the file we want to encrypt with the specified algorithm.

1. Again, I tried to open the encrypted file with **cat** but I saw random characters meaning that the text file was correctly encrypted.
2. Finally, I used **gpg -d** to decrypt with the given password and see the original file:

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Third Part (maximum qualification B): asymmetric encryption**

**Exercise 3**

1. We can observe that we have no keys generated:

A screenshot of a computer

AI-generated content may be incorrect.

1. I used the command **gpg --full-generate-key** to generate the keys:

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

Then I choose:

* Option 1 (RSA and RSA)
* Key size: 4096 bits
* Expiration: 1w (1 week)
* Name: Nicolas
* Emal: ndalessandro@uoc.edu
* Comment: (empty)
* Confirm with **o** and entered the passphrase to protect the private key.

GPG successfully created the public and private keys.

1. I used **gpg –list-keys** to check the generated key. We can see:

**pub** (public key)

**rsa4096** (a 4096-bit RSA key)

Created on May 14, 2025.

**SC** (the key is valid for Signing and Certify)

**E** (expires on May 21, 2025).

The **uid** is Nicolas ndalessandro@uoc.edu

sub (subkey generally used to encrypt E)

Additionally, I ran **gpg –list-private-keys**. This command shows the **secret keys** stored. These are the private keys that match the public keys you have generated. We should **never share** the secret key (It is used to **decrypt messages** and to **sign** them):

**A screenshot of a computer

AI-generated content may be incorrect.**

**Exercise 4**

1. To export the public key, I used this command:

**gpg --output ndalessandro.asc --armor --export ndalessandro@uoc.edu**

A screenshot of a computer screen

AI-generated content may be incorrect.

1. I sent this file to a classmate by email as an attachment.

A screenshot of a computer

AI-generated content may be incorrect.

**Exercise 5**

1. I have imported my colleague public key using the **gpg –import [colleague-file].asc**
2. I verified that the key was imported with the command **gpg --list-keys**
3. I have created an encrypted file for my colleague and used its id to encrypt with it public key.
4. I sent the encrypted file by email.

**Exercise 6**

1. TODO
2. TODO
3. TODO

**Exercise 7**

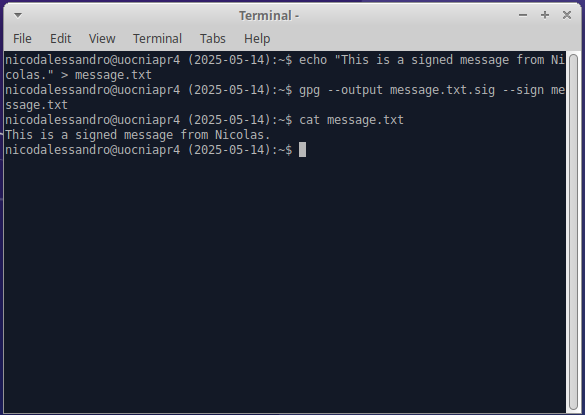
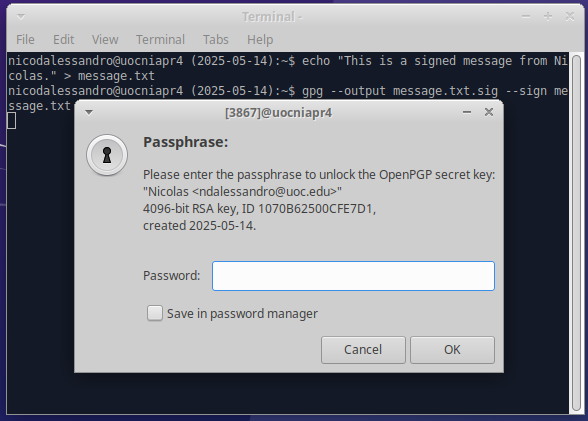
1. To delete a public key from our key ring, we can run the command:

**gpg --delete-key email@address.com**

If we also want to delete the private key, we can run:

**gpg --delete-secret-key email@address.com**

1. To sign a message without encrypting it we can run the command **gpg --output message.txt.sig --sign message.txt**

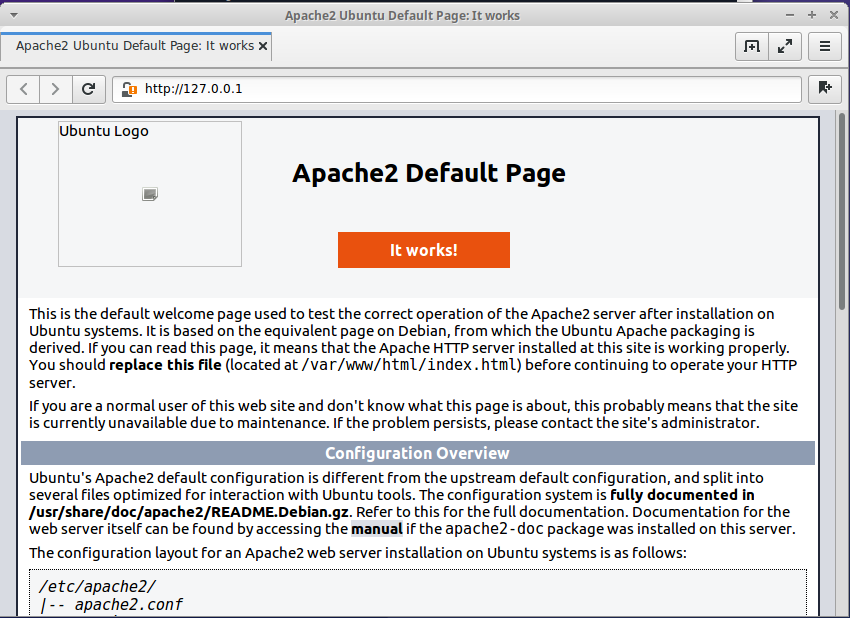
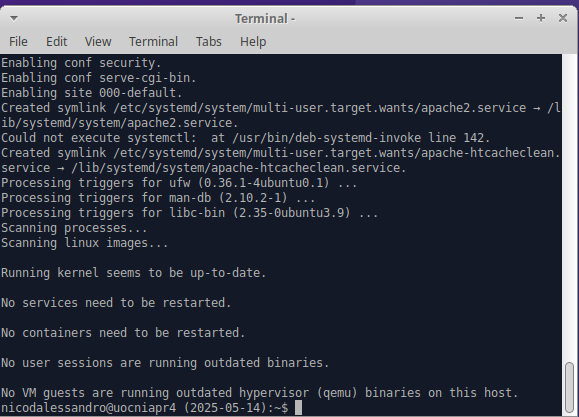


1. It makes sense to sign a message without encrypting because it proves that I am the author and also confirms that the message has not been modified. Even anyone can read the message, they can verify that it is authentic. This is useful for public documents, announcements, or code that must be trusted (but not hidden or encrypted).

**Fourth Part (maximum qualification A): Apache secure configuration, client connection and SSL/TLS explanation**

**Exercise 8**

1. I installed Apache and check it is working:



1. I modified the default page to show the requested message:

A screenshot of a computer

AI-generated content may be incorrect.

1. SSL directives in **default-ssl.conf**:

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

We can observe:

1. **SSLEngine on** - Enables SSL/TLS for this virtual host.
2. **SSLCerificateFile** - Path to the certificate document used to encrypt the connections.
3. **SSLCertificateKeyFile** – Path to the private key that matches the certificate.
4. **<FilesMatch> and <Directory>** - Blocks to define environments and permissions for CGI scripts when using SSL (small programs that run on the web server).
5. Enable HTTPS:

A screenshot of a computer

AI-generated content may be incorrect.

**A screenshot of a computer

AI-generated content may be incorrect.**

1. Confirm with **nestat -ntl** that HTTPS is enabled:

**A computer screen shot of a computer

AI-generated content may be incorrect.**

**Exercise 9**

A screenshot of a computer

AI-generated content may be incorrect.

1. **Protocol (example ﻿Protocol : TLSv1.3)** - This shows the version of the TLS that was used in the connection. As we already know, TLS is the modern name for SSL. The 1.3 version is currently the latest and more secure. This means we are using the latest and more secure encryption protocol in this connection.
2. **Cipher (example Cipher : TLS\_AES\_256\_GCM\_SHA384)** - This shows the full elements of encryption algorithms that were used in this connection. We can see the **AES256** (AES encryption algorithm with a 256-bit key). **GCM** which means Galois/Counter Mode (a secure and fast way to encrypt blocks of data), and finally **SHA384**, which is a hash algorithm that is used for authentication and integrity to detect changes in data.
3. **Session ID (example Session-ID: 1DE87BBD507C…)** - This is the unique ID that was assigned to this session between the client and the server. If the same client connects relatively soon, the ID will allow for session reuse to make the connection faster, since it will avoid repeating the full TLS handshake, saving time and resources.
4. **STLS Session ticket (example 0000 - 76 00 b3 bc c4 14 0a c8 ...)** - This is a block of encrypted data that is being sent by the server. It contains all the session info (including the keys) so the client can resume the session later. It is kind of a saved “cookie” but for TLS session. This component is part of a feature called session resumption, that improves speed and performance in future connections but without losing security.

**Exercise 10**

A screenshot of a computer

AI-generated content may be incorrect.

1. As we can observe, when opening the localhost, the browser shows a “Security unknown” because the certificate was not signed by a trusted Certificate Authority or CA, but instead self-signed by the server (CN = ucniapr4), hence requesting me to trust it manually:

A screenshot of a computer

AI-generated content may be incorrect.

1. When using Wireshark (as we can see in the image above) I filtered the TLS packages and identify the following algorithms for the TLS 1.2 protocol used.

We can observe in the Server Hello response to the Client Hello options, that the algorithm used in the connection was:

**TLS\_ECDHE\_RSA\_WITH\_AES\_128\_GCM\_SHA256**

* Key exchange: ECDHE
* Authentication: RSA
* Encryption: AES 128 in GCM mode
* Integrity: SHA256

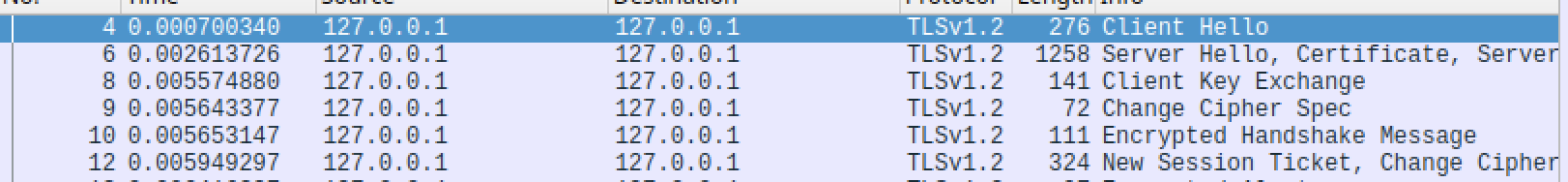
A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

1. We can clearly see in the image the three main steps of the TLS handshake:



1. **Client Hello:** The client says “hello” to the server and send the supported TLS version and cipher suites (Wireshark capture line 4)
2. **Server Hello + Certificate:** The server replies with the selected cipher and the certificate. This certificate includes the public key and the identity (Wireshark line 6)
3. **Key Exchange + Finished**: The client sends the key exchange and change the cipher spec. Then, the server responds with it own encrypted messages. Finally, the encrypted communication begins (Wireshark capture lines 8, 9, 10 and 12).
4. A self-signed certificate offers some protection but not all:
5. What it guarantees (Encryption): The data is private, and it is encrypted between the browser and the server. This means that no one can read it if they are watching the network.
6. What it does not guarantee (Identity - Trust): We don´t know who the owner of the website is, and since anyone can create a certificate with any name, the browser does not trust the certificate (as we already explained is not signed by a trusted company (Certificate Authority).

In summary, even a self-signed certificate may be good for this learning purposes, development or personal servers it won´t be safe for the public without some extra steps.