Ejercicios:

1. **Junto a este documento se adjunta un archivo ej1.txt que contiene ciphertext en base64. Sabemos que el método para encriptarlo ha sido el siguiente:**

   La clave es de 10 caracteres. Frodo ha pensado que era poco práctico implementar one time pad, ya que requiere claves muy largas (tan largas como el mensaje). Por lo que ha decidido usar una clave corta, y repetirla indefinidamente. Es decir el primer byte de cipher es el primer byte del plaintext XOR el primer byte de la clave. Como la clave es de 10 caracteres, sabemos que el carácter 11 del ciphertext también se le ha hecho XOR con el mismo primer carácter de la clave. Debajo vemos un ejemplo donde la clave usada es "ABC".

   ¿Por qué no ha sido una buena idea la que ha tenido Frodo? Explica cómo se puede encontrar la clave y desencriptar el texto a partir del archivo.

   Sabiendo que la clave es de 10 bytes, se divide el ciphertext en bloques de 10 bytes, para luego agrupar todos los primeros, los segundos, los terceros, … Después se realiza una búsqueda con los 128 bytes posibles y se le aplica un análisis de frecuencia de letras (ETAOIN SRHDLU, para el inglés) la que mejor puntuación saque será la clave. Solo queda repetir la clave para todo el ciphertext y Sauron se habría enterado del todo.

   El código se adjunta en **1.py**.

2. **En el archivo ej2.py encontramos dos funciones. Una de ellas obtiene una string cualquiera y genera una cookie encriptada con AES CTR, la función genera cuentas con rol user, y se encarga de verificar que la string no contiene ni ';' ni '='. La otra función desencripta la cookie y busca la string ';admin=true;'. Realizando llamadas a estas dos funciones, genera una cookie y modifica el ciphertext tal que al llamar la función de check te devuelva permisos de admin. (Obviamente se puede acceder a la clave de encriptación de la clase, pero el ejercicio consiste en modificar el sistema tal y como si el código se ejecutara en un servidor remoto al que no tenemos acceso, y solo tuvieramos acceso al código).**

   Sabiendo la composición de la cookie, lo que he hecho ha sido crear otra cookie sustituyendo el prefijo y sufijo por ceros, ya que lo que se hace es XOR eso lo dejara igual. Para conseguir que me sustituya el user_data por los permisos de admin, contando con que **len(;admin=true) = 11**, se necesita crear un usuario con la misma longitud y en la cookie modificada hacer un XOR de **b';admin=true'** y **b'user_data'**. Por último, se realiza XOR del ciphertext de la cookie con la cookie modificada, esto dará como resultado un bit-flipping del usuario a los permisos de admin.

   El código se adjunta en **2.py**.

3. **Explica cómo modificarías el código del ejercicio 2 para protegerte de ataques en los que se modifica el ciphertext para conseguir una cookie admin=true.**

   Para protegerme de dichos ataques, utilizaría un sistema GCM. Usa el mismo sistema de encriptación que CTR, pero además durante la encriptación se le pasa un Authentication Tag. Cuando se procede a desencriptar el ciphertext también se verifica que el Auth. Tag sea correcto, impidiendo así la modificación del ciphertext.

4. **Existe otro error en la implementación del Server que le puede hacer vulnerable a otro tipo de ataques, concretamente en los parámetros que usa para CTR. ¿Sabrías decir cuál es? ¿Cómo lo solucionarías?**

   En la línea 27 del modo CTR, sigue encriptando los datos aun sin encriptar. Al pasarle el chipertext ya encriptado con algo añadido, lo volvería a mater al bucle. Sería vulnerable a ataques de length extensión, añadiendo algo al final lo incluiría en la encriptación. Para solucionarlo, la implementación de un modo HMAC que verifique la longitud e integridad del ciphertext (como GCM).

5. **En el archivo ej5.py encontrarás una implementación de un sistema de autenticación de un servidor usando JWT. El objetivo del ejercicio es encontrar posibles vulnerabilidades en la implementación del servidor, y explicar brevemente cómo podrían ser atacadas.**

   - **La función register crea un nuevo usuario en el sistema (en este caso guardado localmente en el objeto AuthServer en la lista users).**
   - **La función login verifica que el password es correcto para el usuario, y si es así, devuelve un JWT donde sub = user y con expiración 6h después de la creación del token.**
   - **La función verify verifica que la firma es correcta y devuelve el usuario que está autenticado por el token.**

   En el registro, usa el modo de SHA1 para hacer el hash de la password, el cual se considera vulnerable a ataques de colisión. También es mucho mejor utilizar **Salting** al hacer el hash de la password. Esto añade al password una string antes de realizar el hash, dificultando así los ataques de fuerza bruta. O utilizar **PBKDF2**, esto además del salt, se encripta miles de veces el salt y el passsword, haciendo aún más difícil el ataque por fuerza bruta.

   En la verificación, se incluye el algoritmo ‘None’, esto permite que alguien desencripte un token y saber su estructura. Solo habría que modificar el header para que el algoritmo sea ‘None’ y el tipo ‘JWT’ ({“alg”:”None”,”typ”:”JWT”}). El payload se modificaría el usuario para el que va dirigido. La firma, se dejaría vacía.

6. **Modifica el código para solucionar las vulnerabilidades encotradas.**

   El código se adjunta en **5.py.**

7. **¿Qué problema hay con la implementación de AuthServer si el cliente se conecta a él usando http? ¿Cómo lo solucionarías?**
   Si un usuario intenta comunicarse con el servidor por medio de http, se le negara el acceso. Ya que por medio de http no se pude verificar la identidad del cliente.
   Para solucionarlo implementaría https en el servidor, así por medio de un hadshake y un certificado se podría verificar la identidad del cliente.

8. **Sabemos que un hacker ha entrado en el ordenador de Gandalf en el pasado. Gandalf entra en internet, a la página de Facebook. En la barra del navegador puede ver que tiene un certificado SSL válido. ¿Corre algún riesgo si sigue navegando?**

   Si, podría ser una pagina de Facebook con un certificado de baja confianza con alguna pequeña variación en la url (Faceboook, Faccebook, …) que se usase para phishing.
   El hacker también podría haber instalado un navegador modificado en el que hubiese añadido su propio certificado a la lista de confianza de dicho navegador.

9. **En el archivo sergi-pub.asc mi clave pública PGP:**
   - **Comprueba que la fingerprint de mi clave es: DAB01687C2910408227DD51306D51234C8B631A2**
   - **Genera un par de claves, y añade tu clave pública a los archivos entregados, el archivo con tu clave pública tiene que estar encriptado para que lo pueda ver sólo con mi clave privada.**
   - **Usa tu clave privada para firmar el archivo de entrega, y adjunta la firma en un documento firma.sig (Recuerda que deberás hacer la firma con la versión final que entregues, ya que, si realizas algún cambio, la firma no será válida).**

   1. Verifico el fingerpint: **gpg --show-keys Sergi.asc**
   2. Añado la clave a mi keyring: **gpg --import Sergi.asc**
   3. Compruebo que se ha añadido: **gpg --list-keys**
   4. Genero mi clave pública: **gpg --gen-key**
   5. Exporto a un archivo mi clave publica: **gpg --output Adrian.asc --armor --export Adrian**
   6. Encripto el archive, añadiendo la clave de Sergi y mi clave para poder desencriptalo: **gpg --output Adrian.gpg --encrypt --recipient Sergi --recipient Adrian Adrian.asc**
   7. Firmo el PDF de la practica**: gpg --output Practica_Crypto.sig --detach-sign Practica_Crypto.pdf**
   8. Por ultimo verifico la firma: **gpg --verify Practica_Crypto.sig Practica_Crypto.pdf**