Respuesta a ejercicios de la “PRÁCTICA: CRIPTOGRAFÍA”.
Los enunciados completos se encuentran en Práctica Bootcamp 11.pdf
Los scripts utilizados para la resolución de los ejercicios se encuentran en el repositorio de github:

1. ¿Qué valor ha puesto el Key Manager en properties para forzar dicha clave final?
   20553975C31055E

   ¿Qué clave será con la que se trabaje en memoria?
   08653F75D31455C0

   Código:
   https://github.com/ImDavidGR/keepcoding_ciberseguridad/blob/main/ejercicio_1.py

2. ¿qué obtenemos?
   Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.

   ¿Qué ocurre si decidimos cambiar el padding a x923 en el descifrado?
   Obtenemos el mismo resultado:
   Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.

   ¿Cuánto padding se ha añadido en el cifrado?
   El padding es de 1 byte

   En este caso particular, ambos funcionan y dan el mismo resultado.
   El padding es de 1 byte (0x01), que es válido tanto para PKCS7 como X923.
   PKCS7 con 1 byte: 0x01
   X923 con 1 byte: 0x01

   Código:
   https://github.com/ImDavidGR/keepcoding_ciberseguridad/blob/main/ejercicio_2.py

3. ¿Cómo podríamos mejorar de forma sencilla el sistema, de tal forma, que no sólo garantizamos la confidencialidad sino, además, la integridad del mismo?
   ChaCha20 SOLO (Sistema Actual - SOLO CONFIDENCIALIDAD)
   - PROBLEMA: No hay garantía de INTEGRIDAD
     - Un atacante podría modificar el texto cifrado
     - No detectaremos manipulaciones maliciosas
     - Solo tenemos CONFIDENCIALIDAD, no AUTENTICACIÓN

   ChaCha20-Poly1305 (PROPUESTA MEJORADA - CONFIDENCIALIDAD + INTEGRIDAD)
   - Confidencialidad: El texto está cifrado.
   - Integridad: Detecta cualquier modificación del cifrado.
   - Autenticidad: Verifica que el emisor tiene la clave correcta.
   - AAD: Protege metadatos sin cifrarlos.
   - Eficiencia: Un solo paso para cifrar y autenticar.
   - Estándar: RFC 8439, usado en TLS 1.3, WireGuard, Signal.

Código:
https://github.com/ImDavidGR/keepcoding_ciberseguridad/blob/main/ejercicio_3.py

4. ¿Qué algoritmo de firma hemos realizado?
   Algoritmo de firma: HS256

   ¿Cuál es el body del jwt?
   Body del JWT: {'usuario': 'Don Pepito de los palotes', 'rol': 'isNormal', 'iat': 1667933533}

   ¿Qué está intentando realizar?
   El hacker modificó el campo 'rol' de 'isNormal' a: isAdmin

   ¿Qué ocurre si intentamos validarlo con pyjwt?
   Nos da el error InvalidSignatureError. Dado que la firma no es válida y el token ha sido modificado.

   Código:
   https://github.com/ImDavidGR/keepcoding_ciberseguridad/blob/main/ejercicio_4.py

5. ¿Qué tipo de SHA3 hemos generado?
   Tipo de SHA3: SHA3-256

   Y si hacemos un SHA2, y obtenemos el siguiente resultado: [...]
   ¿Qué hash hemos realizado?
   Tipo de SHA2: SHA2-512

   ¿Qué propiedad destacarías del hash, atendiendo a los resultados anteriores?
   Efecto avalancha. Con una sola modificación, como un punto, en la entrada podemos producir un cambio drástico en la salida.

   Código:
   https://github.com/ImDavidGR/keepcoding_ciberseguridad/blob/main/ejercicio_5.py

6. Calcula el hmac-256 del siguiente texto: [...]
   HMAC-SHA256:
   857d5ab916789620f35bcfe6a1a5f4ce98200180cc8549e6ec83f408e8ca0550

   Código:
   https://github.com/ImDavidGR/keepcoding_ciberseguridad/blob/main/ejercicio_6.py

7. Responderé a estas preguntas como una sola con el siguiente informe:
    a. Primera propuesta: SHA-1
        - Actualmente se considera inseguro y obsoleto debido a vulnerabilidades descubiertas en 2005.
        - En 2010 el NIST lo empieza a prohibir para cualquier uso en el área de la criptografía.
        - En 2017 se descubrió un ataque de colisión en contra de este algoritmo.
        - Su velocidad de cálculo es un problema: aunque útil para integridad de datos, facilita ataques de fuerza bruta o diccionario.

        Conclusión: SHA-1 no es recomendable para almacenar contraseñas.

    b. Segunda propuesta: SHA-256
        Más seguro que SHA-1; no se conocen colisiones prácticas. Sin embargo sigue siendo demasiado rápido para contraseñas, permitiendo que un atacante pueda probar millones de combinaciones por segundo.

        Posible fortalecimiento:
        - Salt: Valor aleatorio único por usuario que ayuda a prevenir ataques con tablas rainbow.
        - Iteraciones: Aplicar SHA-256 varias veces para ralentizar los ataques de fuerza bruta.

    c. Mi propuesta:
        Aunque actualmente contamos con SHA-3 como evolución de SHA-2 tenemos otras mejores opciones como bcrypt, scrypt, Argon2, siendo este último el más recomendado actualmente.

        Ventajas:
        - Integran salt automáticamente.
        - Se pueden configurar para que sean más lentos y usen memoria, dificultando así ataques incluso con hardware especializado.

    Código:
https://github.com/ImDavidGR/keepcoding_ciberseguridad/blob/main/ejercicio_7.py

8. Haremos lo mismo que antes. Responderé de una.
    En el escenario propuesto, al tratarse de posibles transacciones con tarjeta bancaria, se requiere un sistema criptográfico robusto y eficiente. Para garantizar la confidencialidad de los campos sensibles, se puede utilizar el algoritmo AES con clave de 256 bits, ya que es un estándar ampliamente utilizado y ofrece un buen rendimiento.

    Sin embargo, AES por sí solo no garantiza la integridad del mensaje, por lo que debe complementarse con un mecanismo de autenticación como HMAC-SHA256, que permita detectar cualquier modificación del contenido. De esta forma se garantiza tanto la confidencialidad como la integridad de los mensajes, incluso en ausencia de TLS.

    Adicionalmente, para evitar ataques de repetición (replay attacks), se puede incluir en cada mensaje un valor único, como un timestamp o un nonce, que será verificado

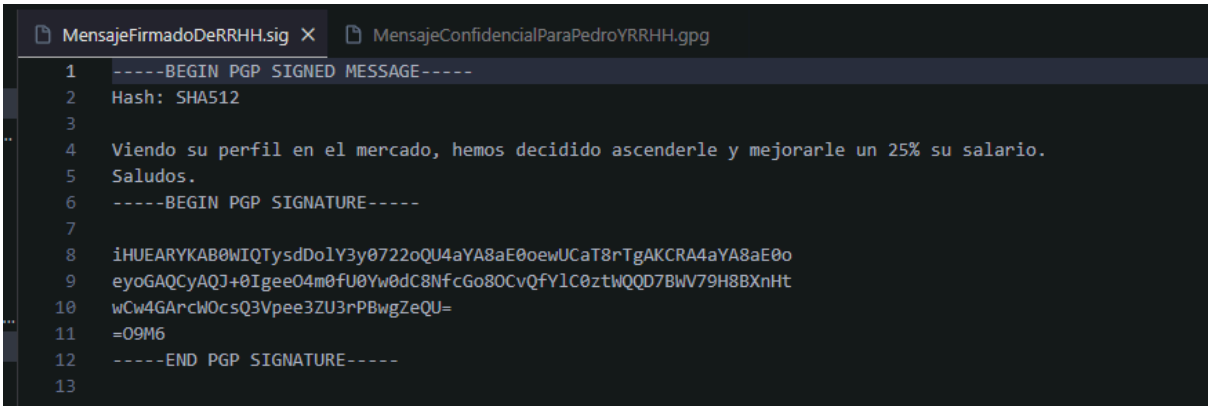
por el servidor. Dicho valor debe formar parte de los datos autenticados, de modo que un mensaje previamente capturado y reenviado sea rechazado por el sistema.

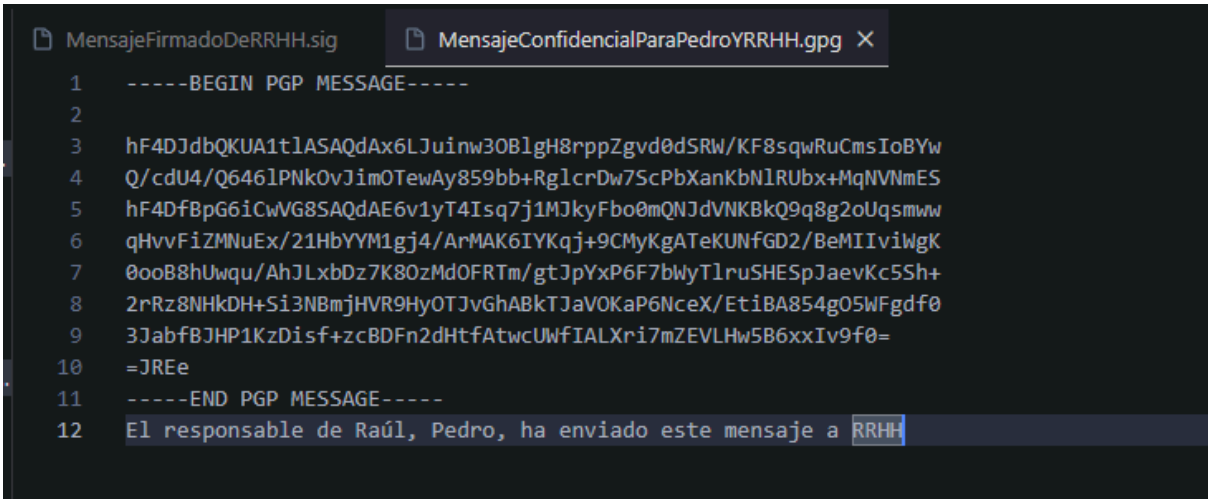
Código:
https://github.com/ImDavidGR/keepcoding_ciberseguridad/blob/main/ejercicio_8.py

9. Se requiere calcular el KCV de las siguiente clave AES: [...]
   KCV SHA-256: DB7DF2
   KCV AES-CBC: 5244DB

   Código:
   https://github.com/ImDavidGR/keepcoding_ciberseguridad/blob/main/ejercicio_9.py

10. El responsable de Raúl, Pedro, ha enviado este mensaje a RRHH [...]
    Lo resolví con Python. Dejaré por aquí la captura de la firma generada. Y al igual que en los demás ejercicios el enlace al código.





    Código:
    https://github.com/ImDavidGR/keepcoding_ciberseguridad/blob/main/ejercicio_10.py

11. ¿Por qué son diferentes los textos cifrados?
    Los textos cifrados son diferentes dado que RSA-OAEP incorpora aleatoriedad en el proceso de padding.

Cada operación de cifrado genera un valor aleatorio único que se mezcla con el mensaje original, garantizando que incluso cifrando el mismo contenido con la misma clave pública, el resultado sea siempre diferente.
Esta es una característica de seguridad fundamental que previene ataques de análisis de patrones.

Aún así el mensaje descifrado (la clave simétrica original) es el mismo en ambos casos.

Código:
https://github.com/ImDavidGR/keepcoding_ciberseguridad/blob/main/ejercicio_11.py

12. ¿Qué estamos haciendo mal?
    1. [GRAVE] Reutilización del Nonce en AES/GCM
       En el enunciado se indica:
       "Nuestro sistema usa los siguientes datos en cada comunicación con el tercero"

       AES/GCM requiere obligatoriamente que el nonce (IV):
       - Sea único para cada cifrado realizado con la misma clave
       - Nunca se reutilice

       Consecuencias de reutilizar el nonce:
       - Un atacante puede recuperar información del texto plano
       - Puede crear mensajes válidos
       - Se pierde tanto confidencialidad como autenticidad

    2. [MODERADO] Longitud y formato incorrecto del nonce
       El nonce proporcionado es:
       9Yccn/f5nJJhAt2S

       El nonce se recomienda que tenga una longitud de 12 bytes (96 bits) para AES/GCM.
       Si el nonce no tiene la longitud adecuada, puede afectar la seguridad del cifrado.

    3. [GRAVE] Falta de mención del Authentication Tag
       No se menciona el Authentication Tag en el enunciado.
       El Authentication Tag es crucial en AES/GCM para garantizar la integridad y autenticidad del mensaje.
       Sin el tag, no se puede verificar si el mensaje ha sido alterado durante la transmisión.

Cifra el siguiente texto: [...]
Texto cifrado en hexadecimal:
5dcbb6261d0fba29ce39431e9a013b34cbca2a4e04bb2d90149d61f4afd04d65e2abd
d9d84bba6eb8307095f5078fbfc16256d6120e37aa4c3ecfd9261640dcc46410d

Texto cifrado en base64:
Xcu2Jh0PuinOOUMemgE7NMvKKk4Euy2QFJ1h9K/QTWXiq92dhLum64MHCV9Qe
Pv8FiVtYSDjeqTD7P2SYWQNzEZBDQ==

Código:
https://github.com/ImDavidGR/keepcoding_ciberseguridad/blob/main/ejercicio_12.py

13. ¿Cuál es el valor de la firma en hexadecimal?
Firma RSA PKCS#1 v1.5 en hexadecimal:
a4606c518e0e2b443255e3626f3f23b77b9d5e1e4d6b3dcf90f7e118d6063950a23885
c6dece92aa3d6eff2a72886b2552be969e11a4b7441bdeadc596c1b94e67a8f941ea99
8ef08b2cb3a925c959bcaae2ca9e6e60f95b989c709b9a0b90a0c69d9eaccd863bc92
4e70450ebbbb87369d721a9ec798fe66308e045417d0a56b86d84b305c555a0e7661
90d1ad0934a1befbbe031853277569f8383846d971d0daf05d023545d274f1bdd4b00
e8954ba39dacc4a0875208f36d3c9207af096ea0f0d3baa752b48545a5d79cce0c2ebb
6ff601d92978a33c1a8a707c1ae1470a09663acb6b9519391b61891bf5e06699aa0a0
dbae21f0aaaa6f9b9d59f41928d

Firma Ed25519 en hexadecimal:
bf32592dc235a26e31e231063a1984bb75ffd9dc5550cf30105911ca4560dab52abb40
e4f7e2d3af828abac1467d95d668a80395e0a71c51798bd54469b7360d

Código:
https://github.com/ImDavidGR/keepcoding_ciberseguridad/blob/main/ejercicio_13.py

14. ¿Qué clave se ha obtenido?
Clave AES:
e716754c67614c53bd9bab176022c952a08e56f07744d6c9edb8c934f52e448a

Código:https://github.com/ImDavidGR/keepcoding_ciberseguridad/blob/main/ejercicio_14.py

15. Antes de responder hagamos un análisis del bloque con
https://paymentcardtools.com/key-block

D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D514ACDB
E6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E03
CD857FD37018E111B

Offset	Version	Value	Meaning
0	Algorithm	D	TR-31 Key Block protected using the AES Key Derivation Binding Method
1-4	Key Block length	0144	Total length of key block
5-6	Key usage	D0	Data Encryption Key (Generic)
7	Algorithm	A	AES
8	Mode of use	B	Both encryption and decryption
9-10	Key Version Number	00	Key versioning is not used for this key
11	Exportability	S	Sensitive, exportable under untrusted key
12-13	Number of optional blocks	00	No optional blocks
14-15	Reserved for future use	00	

Encrypted Key Data (120 bytes)
42766B9265B2DF93AE6E29B58135B77A2F616C8D514ACDBE6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E03CD857FD37

Key Block Authenticator (MAC)
018E111B

a. ¿Con qué algoritmo se ha protegido el bloque de clave?
   AES (el bloque TR-31 está cifrado con AES, byte 7 = A)
b. ¿Para qué algoritmo se ha definido la clave?
   AES (clave de cifrado de datos, byte 5-6 = D0)
c. ¿Para qué modo de uso se ha generado?
   Tanto para cifrar como para descifrar datos (byte 8 = B, "Both")
d. ¿Es exportable?
   Sí, se puede exportar de forma segura bajo clave no confiable (byte 11 = S)
e. ¿Para qué se puede usar la clave?
   Para cifrar y descifrar datos sensibles
f. ¿Qué valor tiene la clave?
   Para obtenerla hay que "desenvolver" (unwrap) el bloque cifrado usando AES Key Wrap con la clave de transporte: A1A10101010101010101010101010102.
   El resultado será la clave de datos real en hexadecimal.

Código:
https://github.com/ImDavidGR/keepcoding_ciberseguridad/blob/main/ejercicio_15.py