**1**. Utilizando la misma fórmula para calcular el XOR a partir de una clave hexadecimal de 16 bytes, se puede averiguar si la clave final en memoria conociendo las primeras dos claves, o inputs. Es decir, si la clave del programador A y la dinámica B al hacer el cálculo dan la clave final en memoria C. El método utilizado en Python lo muestra claramente:

A computer screen shot of numbers

Description automatically generated

*(Ver xor.py)*

Eso sí, la pregunta pide saber el valor que el Key Manager ha utilizado para que salga como clave final 91BA13BA21AABB12. Es decir que en vez de A^B=C queriendo averiguar C, lo que se quiere saber es que es B. Se puede utilizar el mismo método de arriba para saberlo. Simplemente se meten los dos códigos, y nos dará el resultado.

Siendo la clave fija B1EF2ACFE2BAEEFF y la clave final 91BA13BA21AABB12 en memoria, sería 20553975C31055ED.



La segunda pregunta nos pide hacer lo que inicialmente he comentado. Cambiamos el segundo código que es el de producción y dinámico, y nos dará el siguiente resultado:

A screenshot of a computer code

Description automatically generated



**2**. Utilizando la etiqueta de cifrado sim-aes-256 y utilizando la clave encontrada en el keystore, podemos descifrar el texto cifrado “TQ9SOMKc6aFS9SlxhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4USt3aB/i50nvvJbBiG+le1ZhpR84oI=”. Para hacerlo, sabiendo que se ha utilizado AES/CBC/PKCS7, podemos utilizar el siguiente código.

A screen shot of a computer code

Description automatically generated

*(Ver descifrado sencillo.py)*

Al descifrarlo obtenemos el siguiente mensaje:



“Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.”

Si cambiamos el tipo de padding de PKCS7 a x923:

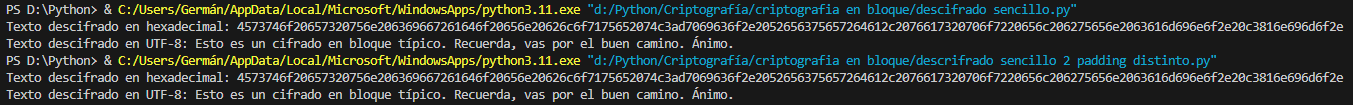
A screenshot of a computer program

Description automatically generated

*(Ver descifrado sencillo 2 padding distinto.py)*

Vemos que no cambia el resultado final en sí.





Los dos resultados tienen la misma longitud. Por lo tanto, no ha habido ningún cambio en el padding.

**3**. Podemos utilizar la programación Python para cifrar la frase “KeepCoding te enseña a codificar y a cifrar”. Para ello, el siguiente código nos da según el nonce proporcionado y la clave del keystore sim chacha 256 el siguiente cifrado:

A computer screen shot of a code

Description automatically generated

*(Ver ChaCha20-Cifrado.py)*

El resultado que nos da es: 

Para mejorar la confidencialidad e integridad, se pueden hacer las siguientes mejoras:

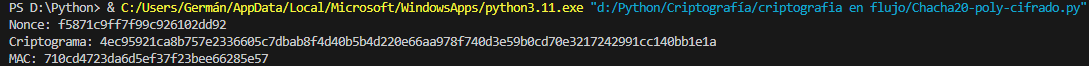
1. No fijar un nonce. Utilizar un nonce aleatorio de 12 bytes. Siempre cabe la posibilidad de que un atacante acabe utilizando un ataque de repetición, y siempre va a ser más vulnerable un sistema donde el atacante pueda descubrir el nonce utilizado y descifrar mensajes.
2. En vez de utilizar el chacha20 tal cual, se puede utilizar el chacha20 Poly, que utilice un único nonce una sola vez, ayudando a crear una transmisión de mensaje único. Además, se puede utilizar un código de autenticación de mensajes, un MAC. Es un valor que se adjunta al cifrado previo. Cualquier intento de cambiar el cifrado, cambia el MAC y por lo tanto se puede detectar cuando no coincide el MAC del receptor y del transmisor.

Utilizando el nonce anterior y el chacha20 Poly, este sería el código: A screen shot of a computer program

Description automatically generated

*(Ver Chacha20-Poly-Keepcodingenseña.py)*

Y nos daría el resultado siguiente:



Ahora vemos que tenemos el cifrado y el MAC. De esta forma, es más difícil alterar el mensaje al igual que más fácil detectar cualquier intento de descifrar o cambiar el mensaje al receptor.

**4**. Utilizando la página de jwt.io para decodificar el JWT, vemos el header, el cuerpo y la firma del texto separados por un punto. Vemos que está codificado con Base64, así que pegamos el JWT en el campo para decodificar.

A screenshot of a computer

Description automatically generated

El resultado que nos da es:

A screenshot of a computer

Description automatically generated

Vemos que se ha utilizado el HS256 para cifrar la firma, el cual representa el HMAC SHA-256.

Para identificar el cuerpo, o el body, es tan sencillo como ver el texto cifrado que se separa por los dos puntos a la izquierda y derecha. Sería esta parte “eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzfQ”. Decodificado, nos muestra lo siguiente:

A white background with purple text

Description automatically generated

El hacker ha intentado hacer una modificación a través de este JWT “eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzQWRtaW4iLCJpYXQiOjE2Njc5MzM1MzN9.krgBkzCBQ5WZ8JnZHuRvmnAZdg4ZMeRNv2CIAODlHRI”. Si lo decodificamos, vemos que lo que ha cambiado es el cuerpo, o payload. Ahora nos muestra esto:

A white background with purple text

Description automatically generated

El hacker está intentando obtener privilegios de administrador a través del usuario “Don Pepito de los palotes”.

Al intentar validarlo utilizando la biblioteca de PyJWT de Python con el siguiente código:

A black background with orange and white lines

Description automatically generated

*(jwt validar.py)*

El código imprimirá el contenido sin problemas si el JWT es válido. Si no, imprime un mensaje diciendo claramente que el token no es válido. El resultado que nos da es:

A black screen with white numbers

Description automatically generated

Esto nos indica que la decodificación ha fallado bien porque la firma no coincide o porque el token se ha cambiado.

**5**. El tipo de SHA3 que se ha generado con el siguiente resultado “bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe” es un SHA3 – 256, por la sencilla razón de que tiene 64 caracteres hexadecimales, representando 32 bytes, el cual es específico de un SHA3-256.

Y si hacemos un SHA2 con el resultado “4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833”, hemos realizado un hash SHA2-512, por que la longitud es de 128 caracteres hexadecimales que representa 64 bytes, el cual es específico al SHA2-512.

Y si generamos con un SHA3-256 utilizando el texto “En KeepCoding aprendemos cómo protegernos con criptografía”, el resultado que nos da es “bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe”, el cual es el mismo resultado que se ofrece en la primera pregunta. Lo importante que hay que destacar es la longitud que tiene el resultado si lo comparamos con el resultado de la segunda pregunta. El hash utilizado es un SHA2-512, por que la longitud es mayor, y por lo tanto representa más bytes. El SHA3-256 son menos bytes representados y por lo tanto tiene una longitud menor.

**6**. Utilizando Python, podemos calcular el HMAC-256 del texto “Siempre existe más de una forma de hacerlo, y más de una solución válida.” Para hacerlo, primero cogemos la clave de Keystore.

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

Con esta clave y texto, utilizamos el siguiente código:

A computer screen with text

Description automatically generated

*(Ver HMAC-gen-reto4.py)*

El código cifra el texto y nos da el siguiente resultado:



“857d5ab916789620f35bcfe6a1a5f4ce98200180cc8549e6ec83f408e8ca0550”.

Es curioso como cambia el resultado, si por ejemplo le quitamos el punto final.





La más mínima modificación de los datos de entrada en el cifrado cambia completamente el cifrado final.

**7.** El SHA-1 se dejó de considerar como un tipo de hash inseguro debido a una serie de vulnerabilidades de colisión que se descubrieron. Este tipo de ataques son serios ya que permiten al atacante utilizar dos datos o mensajes distintos que generan el mismo hash.

El SHA-256 es una mejora en sí del SHA-1, pero para dificultar aún más los ataques de diccionario por ejemplo se podría utilizar para mejorarlo la técnica de “salting” o “salt”. El “salt” sería una cadena aleatoria que se añade o concatena al password antes de hacer el hash. Cada usuario tendría su propio “salt”, cadena aleatoria, y por lo tanto el intentar atacar el hash de otra contraseña no daría los mismos resultados para otras contraseñas.

Si el responsable cree que hay margen de mejora, entonces propondría añadir la técnica del “pepper”. El “pepper”, como el “salt”, es un valor que se concatena con la contraseña, pero a diferencia del “salt” es un valor secreto y que se guarda por separado de los datos del usuario. Entonces el “salt”, que es ‘público’ en el sentido de que se puede averiguar con ciertos ataques, el “pepper” no deja al atacante utilizar ataques de fuerza bruta ya que no tiene ese valor secreto. Aunque se filtrase la base datos con los datos del usuario y el “salt”, les faltaría la segunda parte, el “pepper”.

**9**. Podemos utilizar el siguiente código para calcular el KCV de la clave “A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72”:

A screen shot of a computer program

Description automatically generated

*(Ver KCV(AES).py)*

El código convierte en bytes la clave AES que se da desde el enunciado. Se crea una instancia de SHA256 con la clave y coge los tres primeros bytes de la clave y los muestra en la terminal. En este caso nos da lo siguiente:



“KCV(SHA-256): db7df2”

Creamos después ese bloque de ceros de 16 bytes, junto con el IV también de ceros binarios. Iniciamos el cifrado AES creando una instancia AES con los parámetros en el objeto cipher para después cifrarlo con el bloque de ceros previo. Sacamos los 3 primeros bytes del KCV(AES) y nos da el siguiente resultado:



“KCV(AES): 5244db”

**10**. El responsable de Raúl, Pedro, ha enviado este mensaje a RRHH:

Se debe ascender inmediatamente a Raúl. Es necesario mejorarle sus condiciones económicas un 20% para que se quede con nosotros.

Lo acompaña del siguiente fichero de firma PGP (MensajeRespoDeRaulARRHH.txt.sig). Nosotros, que pertenecemos a RRHH vamos al directorio a recuperar la clave para verificarlo. Tendremos los ficheros Pedro-priv.txt y Pedro-publ.txt, con las claves privada y pública.

Las claves de los ficheros de RRHH son RRHH-priv.txt y RRHH-publ.txt que también se tendrán disponibles.

Se requiere verificar la misma, y evidenciar dicha prueba.

Así mismo, se requiere firmar el siguiente mensaje con la clave correspondiente de las anteriores, simulando que eres personal de RRHH.

Viendo su perfil en el mercado, hemos decidido ascenderle y mejorarle un 25% su salario. Saludos.

Por último, cifra el siguiente mensaje tanto con la clave pública de RRHH como la de Pedro y adjunta el fichero con la práctica.

Estamos todos de acuerdo, el ascenso será el mes que viene, agosto, si no hay sorpresas.

**11**. Utilizamos el código siguiente para recuperar la clave, que sería el siguiente:

A black screen with white text

Description automatically generated

*(Ver RSA\_Descifrado-OAEP.py)*

Este código nos sirve para recuperar la clave utilizando la clave privada que sacamos de un fichero guardado en nuestra carpeta. Se lee la clave y se guarda en keypr. Cogemos el mensaje cifrado, convirtiéndolo de hexadecimal a bytes y lo procesamos por el objeto PKCS1\_OAEP, el cual desciframos con el decryptor. El resultado que nos da es:



“Descifrado: e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72”

Si ahora lo volvemos a cifrar con el mismo algoritmo con el siguiente código:

A screen shot of a computer program

Description automatically generated

*(Ver RSA-Cifrado-OAEPV2.py)*

Y nos da el siguiente resultado:



“Cifrado: ”

Vemos que el cifrado es distinto al original. Esto se debe al relleno que le da el cifrado RSA-OAEP es único y aleatorio. Por lo tanto, aunque el mensaje sea el mismo y el hash utilizado sea igual, el cifrado será siempre distinto cada vez que se haga.

**12**. El uso del algoritmo AES/GCM es bueno, pero tiene un error bastante obvio en los datos utilizados para la comunicación. El nonce parece ser un nonce fijo de longitud de 16 bytes. No es recomendable ni seguro utilizar un nonce fijo para un método de comunicación con algoritmo AES/GCM. Es importante que el nonce en este caso sea siempre único y aleatorio para cada comunicación. El GCM utiliza un contador y el nonce para generar el cifrado único, y por eso si se utiliza un nonce fijo, un atacante puede aprovechar la repetición del nonce para averiguar la clave en sí.

El texto “He descubierto el error y no volveré a hacerlo mal” se puede cifrar utilizando el siguiente código:

A computer screen shot of a program code

Description automatically generated

*(Ver AES-GCM-Cifrado2.py)*

Se convierte de hexadecimal a bytes la clave, y de base64 a bytes el nonce. Se hace el cifrado y se presentan los resultados, los cuales son:



“Texto cifrado en hexadecimal: 5dcbb6261d0fba29ce39431e9a013b34cbca2a4e04bb2d90149d61f4afd04d65e2abdd9d84bba6eb8307095f5078fbfc16256d

Texto cifrado en base64: Xcu2Jh0PuinOOUMemgE7NMvKKk4Euy2QFJ1h9K/QTWXiq92dhLum64MHCV9QePv8FiVt”

**13**. Para saber el valor de la firma en hexadecimal, utilizaremos el siguiente código Python para averiguarlo:

A screen shot of a computer program

Description automatically generated

*(Ver RSASignature-Firmo.py)*

El código importa la clave privada mencionada en la ruta del fichero, convierte en bytes el mensaje mencionado en el enunciado y se hace el hash SHA256 junto con el algoritmo PKCS115, el cual nos da el resultado siguiente:



“Firma ”

Ahora calculamos la firma en hexadecimal con la curva elíptica ed25519, con las claves mencionadas. El siguiente código los puede procesar:

A screen shot of a computer program

Description automatically generated

*(Ver import-ed25519–2.py)*

Importamos las claves mencionadas, las cargamos con las funciones de ed25519 para verificar y firmar y nos aseguramos de que el mensaje este en bytes para después calcular la firma y verificarla. Nos da el siguiente resultado:



“Firma en hexadecimal: bf32592dc235a26e31e231063a1984bb75ffd9dc5550cf30105911ca4560dab52abb40e4f7e2d3af828abac1467d95d668a80395e0a71c51798bd54469b7360d”

**14**. Podemos utilizar el siguiente código para generar esa nueva clave AES usando HKDF con un SHA-512:

A screen shot of a computer code

Description automatically generated

*(Ver HKDF.py)*

Se establece el elemento de diversificación, se proporciona la clave maestra, convirtiéndolos los dos en bytes de hexadecimal. Se hace la derivación con la función HKDF con un SHA-512 para derivar una clave de 32 bytes. El resultado que nos da es:



“Clave key1: e716754c67614c53bd9bab176022c952a08e56f07744d6c9edb8c934f52e448a”

**15**. Utilizando el bloque TR31 indicado y la clave de transporte, los ponemos en el siguiente código:

A black background with colorful lines

Description automatically generated

*(Ver cabeceraTR31.py)*

Nos da los siguientes resultados:

A screen shot of a computer code

Description automatically generated

Vemos que el ID de la clave es D. Es decir, que el bloque está protegido por un AES tal y como indica la documentación (https://github.com/knovichikhin/psec/blob/master/psec/tr31.py):

A screenshot of a computer

Description automatically generated

El algoritmo utilizado se representa con el valor A. Si miramos en la documentación de nuevo, vemos que confirma el ID por igual, es un algoritmo AES:

A screenshot of a computer program

Description automatically generated

Además, se ha generado para un modo de uso con valor B. Si otra vez nos fijamos en la documentación:

A screenshot of a computer program

Description automatically generated

Se utiliza este algoritmo para cifrar y descifrar. La clave también se puede usar según el valor indicado anteriormente como D0. En la documentación nos indica:

A screenshot of a computer

Description automatically generated

La clave se utiliza para el cifrado simétrico, pudiendo llevar el ‘B’, ‘D’ o ‘E’ como modo de uso. Tiene una exportabilidad con valor S, el cual según la documentación nos dice:

A screen shot of a computer

Description automatically generated

El bloque es sensible, pero exportable incluso bajo una clave que no conozcamos.

La clave en sí es útil porque ya previamente está cifrada y eso permite que se pueda compartir de una forma ágil y rápida entre dos personas, por ejemplo.