Práctica Módulo 2 Criptografía

III Bootcamp Full Stack Ciberseguridad KeepCoding

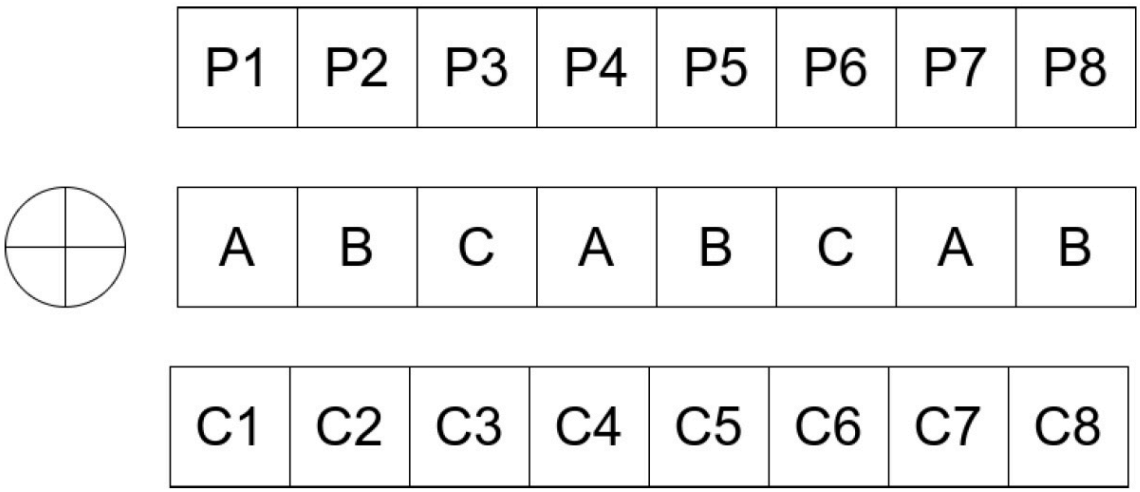
9 de enero de 2022

Marcos Alonso González
alonsogonzalezmarcos@gmail.com
https://github.com/MARALOGON/III-Bootcamp-Full-Stack-Ciberseguridad

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.

Porque si se repiten caracteres en la clave, esta pasa a ser menos segura, más fácil de romper.

Habrá menos combinaciones posibles de la clave, por lo que será más fácil de averiguar haciendo fuerza bruta. Una clave de 10 caracteres diferentes tiene 10^{255} posibles combinaciones diferentes, pero si la longitud del texto que se utiliza en la práctica, tiene pongamos como ejemplo, 1000 caracteres, la clave para ser más segura, debería tener tantos caracteres como tenga el plaintext, en este caso 1000^{255}.

Limitando el número de bytes de la clave se están limitando el número de plaintexts que pueden salir al descifrar el ciphertext, por lo que habrá menos mensajes con sentido.

Programa el código y encuentra la clave junto con el texto.
Pista: no debería ser muy costoso, ya que se puede reutilizar mucho código de un ejercicio hecho en clase ;)

Para encontrar la clave contamos en principio únicamente con un texto cencriptado (ciphertext) en Base64. Dado que sabemos que la clave (key) es de 10 bytes (10 caracteres en ASCII) y que 10 bytes en ASCII equivalen en Base64 a 13,3 caracteres, para facilitar la programación del código, primero vamos a convertir el ciphertext a formato hexadecimal.
De este manera conseguimos que los 10 caracteres de la key en ASCII equivalgan a 20 caracteres del ciphertext en decimal, por lo que la iteración al hacer XOR de los 10 bytes de la clave sobre el ciphertext de manera repetida será más fácil de realizar, al poder dividir el ciphertext en bloques de 20 caracteres y no tener que utilizar paddings como en el caso del ciphertext en Base64.

Convertimos el ciphertext a hexadecimal (Base16):

```
import base64
```

```
mb64_to_mhex = base64.b64decode('Aqui se incluiría el ciphertext en Base64').hex()
```

Lo dividimos en bloques de 20 caracteres:

```
hex_encrypted_message = b"Aqui se incluiría el ciphertext en hexadecimal"
blocks = [ hex_encrypted_message[i:i+20] for i in range(0, len(hex_encrypted_message), 20) ]
```

Sabemos que el plaintext que resulte de hacer key XOR ciphertext está en inglés. Al dividir el ciphertext en bloques de 20 caracteres tenemos 175 bloques de 20 caracteres y un ultimo bloque de 14 caracteres, sobre los que los 10 bytes la key iteraran haciendo XOR para hallar el plaintext.

Dado que no conocemos ningún carácter de los 10 bytes de la key, vamos a utilizar en nuestro código un análisis de frecuencia conocido como "Etaoin shrdlu" basado en los caracteres más repetidos en el idioma inglés. De esta manera, y con un sistema de puntuación adicional también incluido en una función del código, conseguiremos tras realizar la iteración hallar que plaintexts de todos los resultantes cuentan con más probabilidades de ser los correctos, ya que serán los únicos que se impriman al tener menor penalización según el sistema de puntuación implementado en el código, ya que son caracteres más habituales en el idioma inglés.

Para llevar a cabo la iteración XOR de la key con el ciphertext vamos a utilizar lo que se conoce como transposición, con la siguiente función en nuestro código:

```
def transpose(m):
    return [bytes([m[j][i] for j in range(len(m))]) for i in
range(len(m[0]))]
```

Esta función permite que cada byte de la key haga XOR con los 2 caracteres del ciphertext que representan un byte y que le corresponderían por orden. Es decir, el primer carácter de la key (primer byte) hará XOR sobre el primer y segundo caracteres del primer bloque del ciphertext (primer byte), el segundo carácter de la key (segundo byte) hará XOR sobre el tercer y cuarto caracteres del primer bloque del ciphertext (segundo byte), el tercer carácter de la key (tercer byte) hará XOR sobre el quinto y sexto caracteres del primer bloque del ciphertext (tercer byte), y así sucesivamente hasta finalizar el bloque y los 10 bytes de la key. Después, se realizará la misma operación con el segundo bloque del ciphertext, con el tercero, cuarto, etc. hasta el bloque 175. El último bloque (el 176) al contar solo con 14 caracteres hexadecimales y no con 20 como el resto, para XOR solo con los 7 primeros bytes de la key.

Podemos imprimir el resultado y comprobar que el programa habrá generado la key con los 10 caracteres que correspondan al mensaje desencriptado obtenido con menor penalización, es decir, al mensaje que según la función de análisis de frecuencias cuente con más caracteres similares a los caracteres más utilizados en inglés.

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 tuviéramos acceso al código).

El ataque al que es vulnerable CTR es Bit-Flipping. Este es el ataque que debería hacerse para resolver el ejercicio 2.
Como la función check tiene que comprobar que en el plaintext se incluya ‘;admin=true;’ para conceder acceso como admin, pero la función create no permite introducir en el login ni el carácter ; ni tampoco el carácter = , se podría incluir como user_data una string similar a esta

```
user_data = "marcos@gmail.com+admin-true"
```

El siguiente string del plaintext desencriptado, que sería `";safety=veryhigh"` ya incluye un ; al principio, por lo que se haría un ataque de bit-flipping teniendo que modificar únicamente los bits correspondientes a los dos caracteres (+ y -) que hay que cambiar en el ciphertext para poder averiguar que es lo que cambia respecto al ciphertext original.

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.

No he conseguido saber si se podría hacer modificando el código sin usar una HMAC. Se me ocurre que si en el código se puede implementar de alguna manera en la función aes_ctr que si el parámetro data contiene la string ‘;admin=true;’ salte un error. Pero eso haría que el error saltase cuando un usuario que realmente tiene ese rol quiera hacer login.

Así que la solución que encuentro es esta:

Para evitar este tipo de ataques debería utilizarse un Hash-Based Message Authentication Code (HMAC), que protege de ataques de modificación del ciphertext, en los que dentro del tráfico alguien intenta modificar los datos.

HMAC hace un hash de la clave + un hash de la clave y el mensaje encriptado.

$$HMAC(K, m) = H\Big((K \oplus opad) \mid\mid H\big((K \oplus ipad) \mid\mid m\big)\Big)$$

Primero genera un hash del mensaje encriptado + una clave, como se hace con MAC (Message Authentication Code) y luego lo vuelve a hashear haciendo de nuevo XOR con la clave.

MAC es vulnerable a los length-extension attacks, que permiten modificar el mensaje encriptado añadiendo nueva información, y generando un MAC a partir de esa modificación, lo que hace que el servidor dé por valido ese MAC al estar basado en el mensaje encriptado modificado y coincidir con este y no con el mensaje encriptado original.

HMAC evita esta vulnerabilidad al hashear 2 veces, ya que los haches al ser de un tamaño determinado no se pueden extender como si ocurre con el ciphertext.

Python incluye un módulo HMAC* para implementar esta función en cualquier código. En este caso utiliza encriptación md5:

```
import hmac
from hashlib import md5

def hmac_md5(key, msg):
    return hmac.HMAC(key, msg, md5)
```

*Open-source code: https://github.com/python/cpython/blob/main/Lib/hmac.py

Otra forma de evitar esta vulnerabilidad será introducir el modo GCM, también de AES. Este modo utiliza nonces diferentes en cada mensaje encriptado, para evitar utilizar el mismo keystream, y además un authentication tag (similar a HMAC), lo que hace que no se conozca por el momento modo de atacarlo.

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?

Supongo que podría tener que ver con la forma en como se genera el once o la key en la clase, que puede no ser la más segura, pero por más vueltas que le he dado a este ejercicio no he conseguido averiguar a que se refiere el enunciado.

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.

Para JWT usamos la librería PyJWT: https://pyjwt.readthedocs.io/en/latest/
Para funciones criptográficas, usamos la librería PyCryptoDome usada anteriormente en clase.

Guardar los password en su hash equivalente como hace este servidor es una manera más segura de guardar passworrds en una base de datos que guardar el password en si.

Pero esta forma de guardar los hashes de los paswords es vulnerable con fuerza bruta para aquellos passwords más comunes y menos complejos.
Los hashes al ser deterministas se pueden guardar en diccionarios o tablas y se puede por tanto averiguar fácilmente los password menos complejos con programas de crackeo de contraseñas (como John The Ripper o Hashcat) si se consiguiera acceder a la información de la base de datos.

Para solucionar esta vulnerabilidad de las hashes de los passwords menos complejos de romper se puede utilizar el salting. En lugar de hashear solo el password se hashean

juntos un factor aleatorio (salt), que puede ser por ejemplo un número de bytes generados aleatoriamente y el password, haciéndo el hash resultante seguro.
Como el salt se genera aleatoriamente cada vez, aunque 2 personas tengan un password poco complejo, el hash no será nunca el mismo para las 2, por tanto las hash tables son inservibles.

Otra opción para solucionar esta vulnerabilidad es implementar el uso de key derivated functions, que no son vulnerables a hash tables y están diseñados para dificultar la fuerza bruta.
También funciona con salt y password. Ejecuta su función de hasheo varias veces, hasheando los propios hashes sucesivamente (por ej. 5000 a 10000 iteraciones recomendadas en PBKDF2, protocolo de creación de hashes), lo que provoca que la fuerza bruta sea mucho más lenta.

6. Modifica el código para solucionar las vulnerabilidades encontradas.

Para solucionar la vulnerabilidad de los hashes guardados por el servidor, se puede implementar en la función register una variable que funcione como password hasheado con key derivated function, tal como se haría en estas lineas:

```
salt = get_random_bytes(16)

password_hashed = PBKDF2(password, salt, 64, count=10000, 
hmac_hash_module=SHA512)
```

También habría que asegurarse que el servidor no acepte como algoritmo “None”, ya que en ese caso podría generarse cualquier JWT con los datos que se quisiera y pasarlos para que los validara el servidor. En este caso, podríamos generar un JWT que incluyera en el payload rol de administrador y engañar al servidor para que nos diera acceso como admin.

7. ¿Qué problema hay con la implementación de AuthServer si el cliente se conecta a él usando http? ¿Cómo lo solucionarías?

El problema del protocolo http es que es vulnerable a ataques “Man in the middle” en que un tercero puede interceptar las comunicaciones del cliente y hacerse pasar por el servidor. Si ese tercero intercepta la primera comunicación cliente-servidor, el “handshake”, el resto de las comunicaciones ya no tienen ninguna seguridad.

Se soluciona implementando en la comunicación cliente-servidor protocolos de comunicación encriptada, como ssh o https.

El protocolo ssh utiliza un certificado que requiere de conocer la clave publica del servidor para conectarse a el. A través del fingerprint el cliente podrá verificar manualmente por otro canal diferente que se está conectando con el servidor y no con un tercero que he interceptado el mensaje y proporcionado su propia clave pública.

En el protocolo https se comparte una clave inicial entre cliente y servidor cuando se realiza el handshake. El cliente envía una clave publica y el servidor le devuelve una clave privada relacionada con la clave pública recibida. Para verificar que la clave recibida proviene del servidor y no de un tercero que ha interceptado la comunicación, se utilizan los certificados SSL/TLS, que contienen la clave publica del servidor, su dominio, una fecha de expiración y una firma aprobada por un Certificate Authority.

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 puede correr riesgos, ya que aunque en la barra del navegador aparezca símbolo indicando que esa url tiene certificado SSL válido, este puede ser de baja confianza y haber sido generado de forra gratuita sin verificaciones previas (por ej. a través de Let´s Encrypt), para por ejemplo, una pagina de phishing que imite a una página de Facebook.

Para asegurarnos que no se trata de una web de phishing, se puede:

- comprobar que la url es correcta y no engañosa (por ejemplo, es igual que la original y cambia solo un carácter),
- comprobar la información del certificado, lo que se puede hacer a través del navegador, donde se puede ver la cadena de confianza de ese certificado.

9. En el archivo sergi-pub.asc mi clave pública PGP:

• Comprueba que la fingerprint de mi clave es: DAB01687C2910408227DD51306D51234C8B631A2

Para hacer esta comprobación, primero importo en mi maquina la clave publica de Sergi.asc con el comando

```
gpg --import Sergi.asc
```

Haciendo `gpg -list-keys` se comprueba que el fingerprint de la clave publica de Sergi.asc es el mismo que dice el enunciado

```
pub   ed25519 2021-12-14 [SC] [expires: 2026-12-13]
      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.

Genero las claves con el comando

```
gpg --expert -full-gen-key
```

Selecciono la opción de encriptación de curva elíptica

```
(9) ECC and ECC
```

El tipo de encriptación

```
(1) Curve 25519
```

Selecciono el tiempo de expiración de la clave

```
Key is valid for? (0) 1y
```

Indico un nombre para la clave

```
Real name Modulo Criptografía key
```

Le indico una contraseña

```
criptokey
```

Se muestra el fingerprint de la clave pública

```
pub
      85FD5AD1D87F0387A16812440FC86B9BB71CE5D1
```

Con `gpg –list-keys` nos muestra las claves que tenemos guardadas

Para ver la clave publica en base64 que corresponde a la key creada se lanza el comando

```
gpg –armor –export <nombre de la clave> o <fingerprint>
```

El resultado para mi key es:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mDMEYdlKsBYJKwYBBAHaRw8BAQdAjRlNhr+M/GEP5oyJS3haHMQKUfyTaiapz+TG
gndND6m0F01vZHVsbyBDcmlwdG9ncmFmaWEga2V5iJYEExYIAD4WIQSF/VrR2H8D
h6FoEkQPyGubtxzl0QUCYdlKsAIbAwUJAeEzgAULCQgHAgYVCgkICwIEFgIDAQIe
AQIXgAAKCRAPyGubtxzl0TfJAQCz0j4zVcyjHJmAMEGNHOLAicMPEqqb6MGJxjsu
u1/PiAEA0WlPCIzyYP2TDMOFGkdpIfU4z4/kVbpau0lBBQx4OQW4OARh2UqwEgor
BgEEAZdVAQUBAQdAO6U6pnsdKUKpMoErgMU9mDRGhGelIZRwSpbDoIOqWm8DAQgH
iH4EGBYIACYWIQSF/VrR2H8Dh6FoEkQPyGubtxzl0QUCYdlKsAIbDAUJAeEzgAAK
CRAPyGubtxzl0TslAPsFyCZNFAv7wvP6ajaeTmnS7pvZKjnyl3RYt/hw0hGuGAEA
8+nn5noIqs52Oq3rOEJOasjD5pD7hEUkDOgzWmb6DwU=
=Nsd+
-----END PGP PUBLIC KEY BLOCK—
```

Para crear un archivo que contenga la clave pública, lo hacemos con el comando

```
gpg --armor --export Modulo Criptografia key > clave-publica
```

Para obtener la clave privada a un archivo, se hace con el comando

```
gpg --armor -export-secret-key <nombre de la clave> o
<fingerprint>
```

Se introduce la contraseña y se muestra la 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.

Para firmar un archivo de cualquier tipo se debe utilizar el comando

```
gpg --output <nombre del archivo>.sig -detach-sign <nombre del
archivo>
```

Explica los pasos que has seguido.