Práctica Criptografía

KeepCoding

Alejandro Parrado Di Doménico

17 - 29 de Julio de 2022

Ejercicios y Desarrollo

Ejercicio 1

- Tenemos un sistema que usa claves de 16 bytes. Por razones de seguridad vamos a proteger la clave de tal forma que ninguna persona tenga acceso directamente a la clave. Por ello, vamos a realizar un proceso de disociación de la misma, en el cuál tendremos, una clave fija en código, la cual, sólo el desarrollador tendrá acceso, y otra parte en un fichero de propiedades que rellenará el Key Manager. La clave final se generará por código, realizando un XOR entre la que se encuentra en el properties y en el código.

Solución:

Para solucionar el ejercicio se programó un script de Python que se puede encontrar en la ruta ***"01/ejercicio_01.py".*** Este script hace lo siguiente:

1. Define una función **`xor_data()`** encargada de operar dos binarios con XOR.

2. Se define una variable **`k1`** que es la clave en bytes del desarrollador, a la que solo este debe tener acceso.

3. Se obtiene la clave del Key Manager del fichero en la ruta **`01/properties.txt`** y se almacena en la variable **`k2`**.

4. Se hace el cálculo de la clave final a partir de hacer una operación XOR entre **`k1`** y **`k2`**.

- La clave fija en código es A1EF2ABFE1AAEEFF, mientras que en desarrollo sabemos que la clave final (en memoria) es B1AA12BA21AABB12. ¿Qué valor ha puesto el Key Manager en properties para forzar dicha clave final?

Solución:

Se puede solucionar este ejercicio mediante código o mediante una calculadora, en este caso por simplicidad se utilizó la calculadora online [XOR Calculator.]

1. Teniendo en cuenta que ***K = K1 ^ K2***, suponiendo que K es la clave final, K1 la clave del desarrollador y K2 la del Key Manager, entonces la forma de hallarla sería ***K2 = K ^ K1***

2. Al reemplazar los valores de la siguiente forma: ***K2 = B1AA12BA21AABB12 ^ A1EF2ABFE1AAEEFF***, obtenemos que ***K2 = 10453805c00055ed***.

- La clave fija, recordemos es A1EF2ABFE1AAEEFF, mientras que en producción sabemos que la parte dinámica que se modifica en los ficheros de propiedades es B98A15BA31AEBB22. ¿Qué clave será con la que se trabaje en memoria?

Solución:

Siguiendo el hilo de los anteriores ejercicios, tomaremos la clave en memoria (final) como K

1. De tal manera que ***K = A1EF2ABFE1AAEEFF ^ B98A15BA31AEBB22***.

2. Obteniendo que ***K = 18653f05d00455dd***

Ejercicio 2

- Dada la clave con etiqueta **"cifrado-sim-aes-256"** que contiene el keystore. El iv estará compuesto por ceros binarios (**"00"**). Se requiere obtener el dato en claro correspondiente al siguiente dato cifrado:

  zcFJxR1fzaBj+gVWFRAah1N2wv+G2P01ifrKejICaGpQkPnZMiexn3WXlGYX5WnNIosyKfkNKG9GGSgG1awaZg==

- Para este caso, se ha usado un AES/CBC/PKCS7. Si lo desciframos, ¿qué obtenemos?
- ¿Qué ocurre si decidimos cambiar el padding a x923 en el descifrado?
- ¿Cuánto padding se ha añadido en el cifrado?

Solución:

Para solucionar el anterior ejercicio se desarrollaron dos scripts en python ubicados en **`02/`**. El primero **`02/obteniendo_clave_keystore.py`** tiene la función de acceder a las claves almacenadas en el fichero **`02/KeyStorePracticas`** y guardarlas en **`02/keys_output.txt`**.

El segundo script **`02/descifrado.py`** se encarga de acceder al fichero txt con las claves y obtener la que tiene la etiqueta **"cifrado-sim-aes-256"** y luego utilizarla para hacer el descifrado, con un iv de ceros binarios y con el estilo de padding *PKCS7*.

Al ejecutar los scripts obtenemos el mensaje descifrado:

Esto es un cifrado en bloque típico. Recuerda el padding.

Ya que nos lo recuerdan, revisamos el padding que ha sido añadido quitando la función unpad() y observamos que se han añadido 6 bits:

```
696e672e060606060606
```

Ahora si cambiamos el estilo del padding a *x923* obtenemos el siguiente resultado

```
    raise ValueError("ANSI X.923 padding is incorrect.")
ValueError: ANSI X.923 padding is incorrect.
```

Ejercicio 3

- Se requiere cifrar el texto “Este curso es de lo mejor que podemos encontrar en el mercado”. La clave para ello, tiene la etiqueta en el keyStore “cifrado-sim-chacha-256”.
- El nonce “9Yccn/f5nJJhAt2S”.
- ¿Cómo podríamos mejorar de forma sencilla el sistema, de tal forma, que no sólo garanticemos la confidencialidad sino, además, la integridad del mismo?

Solución:

La mejor forma de mejorar el sistema de cifrado en flujo con *ChaCha20* es añadiendo *Poly1305*, de esta forma gracias al tag generado con los datos de autenticación es posible verificar tanto la integridad como la autenticidad del mensaje cifrado.

La solución se encuentra en el fichero **`03/cifrado.py`**, donde nuevamente se obtuvo la clave del Keystore, se almacenó en un txt y luego se obtuvo con el script, con el fin de automatizar el proceso lo máximo posible.

Al hacer un encoding en Base 64 del mensaje cifrado se obtuvo la siguiente cadena:

```
Svcdqsq8DLQ3YC/AHITCT5fB9ivGqCgNARvQAQpdxRuPXXLhTtBJjmDMxxKFEEDH/MFp9JY6D3rYsxShsw==
```

Ejercicio 4

Tenemos el siguiente jwt, cuya clave es “KeepCoding”.

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRmVsaXBlIFJvZHJcdTAwZWRndWV6Iiwicm9sIjoiaXNOb3JtYWwifQ.-KiAA8cjkamjwRUiNVHgGeJU8k2wiErdxQP_iFXumM8
```

- ¿Qué algoritmo de firma hemos realizado?
- ¿Cuál es el body del jwt?

Solución:

Utilizando la herramienta jwt.io se ha reconocido la información.

El body o payload del jwt es:

```
{
  "usuario": "Felipe Rodríguez",
  "rol": "isNormal"
}
```

Se ha utilizado el algoritmo de firma *HMACSHA256*

Al validar la firma con la clave “KeepCoding” verificamos que es **correcta**.

Un hacker está enviando a nuestro sistema el siguiente jwt:

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRmVsaXBlIFJvZHJcdTAwZWRndWV6Iiwicm9sIjoiaXNBZG1pbiJ9.-KiAA8cjkamjwRUiNVHgGeJU8k2wiErdxQP_iFXumM8

- ¿Qué está intentando realizar?
- ¿Qué ocurre si intentamos validarlo con pyjwt?

Solución:

Teniendo en cuenta que el payload del jwt enviado por el hacker es este:

```
{
  "usuario": "Felipe Rodríguez",
  "rol": "isAdmin"
}
```

Podemos afirmar que el hacker está intentando hacer un escalado de permisos, intentando obtener el rol de Admin cambiandolo en la petición.

Al intentar validar este jwt con pyjwt obtenemos el siguiente resultado:

```
jwt.exceptions.InvalidSignatureError: Signature verification failed
```

Ha **fallado** la validación de la firma, es decir que el jwt ha sido alterado y no se puede confiar en su autenticidad ni integridad.

Ejercicio 5

- El siguiente hash se corresponde con un SHA3 Keccak del texto “En KeepCoding aprendemos cómo protegernos con criptografía”.

  bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe

- ¿Qué tipo de SHA3 hemos generado?

Solución:

- Por su longitud (64 dígitos en hexadecimal, es decir 256 bits) es posible saber que se generó un hash de tipo *SHA3-256*

- Y si hacemos un *SHA2*, y obtenemos el siguiente resultado:

  4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833

- ¿Qué hash hemos realizado?

Solución:

Por su longitud sabemos que es un *SHA2-512*

- Genera ahora un SHA3 Keccak de 256 bits con el siguiente texto: “En KeepCoding aprendemos cómo protegernos con criptografía.” ¿Qué propiedad destacarías del hash, atendiendo a los resultados anteriores?

Solución:

Para generar el hash *SHA3-256 Keccak*, se ha desarrollado un script de Python en la ruta **`05/SHA3-256-Keccak.py`** el cual retornó el siguiente hash:

3411df16137dcf332a3aa0cca9107ad448fb330b9f77d617c274ad6840953629

Se puede resaltar que este Hash (keccak) es de la familia **SHA3** el cual es más fuerte por su longitud y por su sistema de “construcción en esponja”.

Referencia: [Difference between SHA-256 and Keccak]

Ejercicio 6

- Calcula el hmac-256 (usando la clave contenida en el KeyStore) del siguiente texto: Siempre existe más de una forma de hacerlo, y más de una solución válida.

Solución:

El script que calcula el hmac de la cadena se encuentra en **`06/HMAC-256.py`**. Este nos da el siguiente mac:

8d3ff74327949f78dffabd403d086ab23a8576a3831a7f28329d52d4c1b0b785

Y al hacer la verificación de integridad del mensaje obtenemos el siguiente mensaje que nos indica que todo está **correcto**:

Mensaje validado ok

Ejercicio 7

- Trabajamos en una empresa de desarrollo que tiene una aplicación web, la cual requiere un login y trabajar con passwords. Nos preguntan qué mecanismo de almacenamiento de las mismas proponemos.
- Tras realizar un análisis, el analista de seguridad propone un hash SHA-1. Su responsable, le indica que es una mala opción. ¿Por qué crees que es una mala opción?

Solución:

Es una mala opción ya que el SHA-1 está roto, aunque se sigue usando ampliamente ya se han encontrado múltiples colisiones, además se tienen mejores versiones como el SHA-2 y el Keccak.

- Después de meditarlo, propone almacenarlo con un SHA-256, y su responsable le pregunta si no lo va a fortalecer de alguna forma. ¿Qué se te ocurre?

Solución:

Yo plantearía la posibilidad de almacenar un HMAC, de esta forma no sólo ocultamos las contraseñas sino que a su vez podemos validar que no haya sido afectada su integridad y autenticidad.

- Parece que el responsable se ha quedado conforme, tras mejorar la propuesta del SHA-256, no obstante, hay margen de mejora. ¿Qué propondrías?

Solución:

Propondría añadir un SALT y un PEPPER al sistema de gestión de contraseñas, de forma que se implementen dos capas más de seguridad, evitando que se comprometan totalmente con un cálculo de rainbow tables, y además que aún accediendo a la base de datos, tenemos como "segundo factor" el pepper posiblemente en un HSM.

Ejercicio 8

- Tenemos la siguiente API REST, muy simple.

Request:

Post /movimientos

Campo	Tipo	Requiere Confidencialidad	Observaciones
idUsuario	Number	N	Identificador
Usuario	String	S	Nombre y Apellidos
Tarjeta	Number	S	

Petición de ejemplo que se desea enviar:

{"idUsuario": 1, "usuario": "José Manuel Barrio Barrio", "tarjeta": 4231212345676891}

Response:

Campo	Tipo	Requiere Confidencialidad	Observaciones
idUsuario	Number	N	Identificador
movTarjeta	Array	S	Formato del ejemplo
Saldo	Number	S	Tendra formato 12300 para indicar 123.00
Moneda	String	N	EUR, DOLLAR


```
{
        "idUsuario": 1,
        "movTarjeta": [{
                "id": 1,
                "comercio": "Comercio Juan",
                "importe": 5000
        }, {
                "id": 2,
                "comercio": "Rest Paquito",
                "importe": 6000
        }],
        "Moneda": "EUR",
        "Saldo": 23400

}
```


- Como se puede ver en el API, tenemos ciertos parámetros que deben mantenerse confidenciales. Así mismo, nos gustaría que nadie nos modificase el mensaje sin que nos enterásemos. Se requiere una redefinición de dicha API para garantizar la integridad y la confidencialidad de los mensajes. Se debe asumir que el sistema end to end no usa TLS entre todos los puntos.
- ¿Qué algoritmos usarías? ¿Serías capaz de hacer un ejemplo en Python de cómo resolverlo?

Solución:

Con base en el objetivo de preservar la confidencialidad e integridad de los mensajes, elegiría un cifrado AES con el modo GCM, ya que este además de permitirnos ocultar la información (en este caso los campos: Usuario, Tarjeta, movTarjeta y Saldo) también podemos verificar el MAC generado mediante el "tag" al enviar la información para validar la integridad de los datos, que no se hayan hecho cambios maliciosos y finalmente poder enviar el response con el mismo cifrado.

No sabría cómo representarlo adecuadamente en un script de Python.

Ejercicio 9

- 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.

Solución:

Al hacer la validación de la firma obtenemos el siguiente resultado:

```
┌──(alejop㉿alejop)-[~/…/courses/KeepCoding/criptografia/Práctica]
└─$ gpg --verify MensajeRespoDeRaulARRHH.txt.sig
gpg: Signature made Sun 26 Jun 2022 06:47:01 AM -05
gpg:                using EDDSA key 1BDE635E4EAE6E68DFAD2F7CD730BE196E466101
gpg:                issuer "pedro.pedrito.pedro@empresa.com"
gpg: Good signature from "Pedro Pedrito Pedro <pedro.pedrito.pedro@empresa.com>"
```

Lo cual nos confirma la **validez** y **autenticidad** del mensaje.

- 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.

Solución:

El fichero firmado con la clave privada de RRHH con el mensaje anterior se encuentra en la ruta **`09/mensaje_para_pedro.sig`**

- 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.

Solución:

El mensaje cifrado se encuentra en **`09/informacion_ascenso_pedro.gpg`**

Ejercicio 10

- Nuestra compañía tiene un contrato con una empresa que nos da un servicio de almacenamiento de información de videollamadas. Para lo cual, la misma nos envía la clave simétrica de cada videollamada cifrada usando un RSA-OAEP. El hash que usa el algoritmo interno es un SHA-256.
- El texto cifrado es el siguiente:

  7edee3ec0b808c440078d63ee65b17e85f0c1adbc0da1b7fa842f24fb06b332c1560
  38062d9daa8ccfe83bace1dca475cfb7757f1f6446840044fe698a631fe882e1a6fc
  00a2de30025e9dcc76e74f9d9d721e9664a6319eaa59dc9011bfc624d2a63eb0e449
  ed4471ff06c9a303465d0a50ae0a8e5418a1d12e9392faaaf9d4046aa16e424ae1e2
  6844bcf4abc4f8413961396f2ef9ffcd432928d428c2a23fb85b497d89190e3cfa49
  6b6016cd32e816336cad7784989af89ff853a3acd796813eade65ca3a10bbf58c621
  5fdf26ce061d19b39670481d03b51bb0eecc926c9d6e9cb05ba56082a899f9aa72f9
  4c158e56335c5594fcc7f8f301ac1e15a938

- Las claves pública y privada las tenemos en los ficheros clave-rsa-oaep-publ.pem y clave-rsa-oaep-priv.pem.
- Si has recuperado la clave, vuelve a cifrarla con el mismo algoritmo. ¿Por qué son diferentes los textos cifrados?

Solución:

Para obtener la clave se desarrolló el script ubicado en 10/RSA_OAEP.py en el cual se importan las claves clave-rsa-oaep-publ.pem y clave-rsa-oaep-priv.pem y se usa la privada para descifrar el mensaje, obteniendo así la siguiente clave:

e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72

Ahora, al volverla a cifrar se obtiene la siguiente cadena hexadecimal:

434ebefaac37c0d6343eed103d99236790ee70d5443e47f8ba96c92b19c2856a6373cf98855b3ce
9b2cab8451019532ecc3bf5f9b4c6d073c184c7b1b5eb372fa69272e0a605b58a9657a11b8c3e65
e40400ed705a9c69e7d93f38710af0d9dd6883d605bda030db54c83a10acf5cb2fc4031afd1e66b
9026a2637726d4b2e2ac1e7319afecafdeac8ae3261265a106bc1ac987b4b289bba401c54a3a28f
a3cb395c66bedd1bbdebcd2ffce04cfd3af05365ced1a23d3fa5f771d40362cbe3e43023a438a3c8

acda7b32bbfae2ac2df13ba4bf14526383025b5a9b57a7d60832342d265c2f08f614e12579ff0968
4213ede4dd75c7589d19bf3b408267d9e5f0

Pero efectivamente, cada vez que se ejecuta el cifrado (el script se encuentra en 10/encrypt.py) se obtiene una cadena distinta. Esto sucede porque no estamos aplicando solo el RSA sino el RSA con un padding OAEP (Optimal Asymmetric Encryption Padding) el cual genera cadenas aleatorias y hace un proceso de XOR entre estas y el mensaje, en el proceso de descifrado se hace lo opuesto, el algoritmo encuentra esta cadena y hace el proceso inverso para hallar el mensaje en claro. Este método se hace con el fin de que un atacante no pueda reconocer una cadena cifrada porque siempre va a ser distinta así se utilice la misma clave pública.

Referencias:

- Why does RSA encrypted text give me different results for the same text
- Optimal Asymmetric Encryption Padding
- RSA (cryptosystem)

Ejercicio 11

- Nos debemos comunicar con una empresa, para lo cual, hemos decidido usar un algoritmo como el AES/GCM en la comunicación. Nuestro sistema, usa los siguientes datos en cada comunicación con el tercero:

  Key: E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB72

  Nonce:9Yccn/f5nJJhAt2S

- ¿Qué estamos haciendo mal?

Solución:

Aquí hay un gran error de implementación AES con el modo GCM, ya que nunca se debe usar el mismo nonce para diferentes mensajes cuando se cifra y descifra en el modo GCM, a diferencia por ejemplo del CBC, por esto es recomendable que el nonce sea un contador que llevan los dos sistemas que se están comunicando, y está totalmente contra-recomendado el uso de un valor de nonce fijo, esto supondría que un atacante pudiera descifrar no uno, sino todos los mensajes enviados mediante este canal de comunicación.