Alejandro García Gutiérrez

Criptografia

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.

La clave fija en código es B1EF2ACFE2BAEEFF, mientras que en desarrollo sabemos que la clave final (en memoria) es 91BA13BA21AABB12. ¿Qué valor ha puesto el Key Manager en properties para forzar dicha clave final?

1. La clave dinámica que puso el Key Manager en desarrollo es: **20553975C31055ED**.

La clave fija, recordemos es B1EF2ACFE2BAEEFF, mientras que en producción sabemos que la parte dinámica que se modifica en los ficheros de propiedades es B98A15BA31AEBB3F. ¿Qué clave será con la que se trabaje en memoria?

1. La clave con la que se trabajará en memoria en producción es: **08653FD3D31455C0**.

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

TQ9SOMKc6aFS9SlxhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4USt3aB/i50nvvJbBiG+le1ZhpR84oI=

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?

1. **¿Qué obtenemos?** Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.
2. **¿Qué ocurre si decidimos cambiar el padding a x923 en el descifrado?**

**Implicaciones al cambiar a X.923:**

**Compatibilidad con el mensaje cifrado**:

* 1. Si el mensaje cifrado fue originalmente rellenado con **PKCS7**, al intentar descifrarlo usando **X.923**, el proceso de unpad fallará.
  2. Esto ocurre porque **X.923** espera ceros en el padding, pero en **PKCS7**, los bytes de relleno no necesariamente son ceros.

**Error específico**:

* 1. Python generará un ValueError del tipo *"Padding is incorrect."* durante el descifrado si los bytes de relleno no coinciden con las expectativas de **X.923**.

1. Padding añadido:
   1. Cantidad de padding añadido: 1 bytes

Bytes de padding: b'\x01'

3 – Se requiere cifrar el texto “KeepCoding te enseña a codificar y a cifrar”. La clave para ello, tiene la etiqueta en el Keystore “cifrado-sim-chacha-256”. El nonce “9Yccn/f5nJJhAt2S”. El algoritmo que se debe usar es un Chacha20.

**texto\_cifrado:** "TslZIcqLdX4jNmBcfbq49NQLW00iDmaql490DT5ZsM1w4yFyQpkcwUC7Hho="

¿Cómo podríamos mejorar de forma sencilla el sistema, de tal forma, que no sólo garanticemos la confidencialidad sino, además, la integridad del mismo? Se requiere obtener el dato cifrado, demuestra, tu propuesta por código, así como añadir los datos necesarios para evaluar tu propuesta de mejora.

Para garantizar **confidencialidad** y **integridad**, podemos utilizar un esquema de cifrado autenticado como **ChaCha20-Poly1305**, que ya garantiza ambos objetivos en su implementación. Sin embargo, para mejorar la seguridad del sistema y evitar problemas con reutilización del **nonce** o errores humanos, podemos implementar un mecanismo que:

1. **Nonce único**:
   * Se genera un **nonce aleatorio** de 12 bytes (get\_random\_bytes(12)) para evitar la reutilización accidental que comprometería la seguridad.
2. **Datos asociados (AAD)**:
   * Agregamos metadatos como UsuarioID y una marca de tiempo (time.time()) en el cifrado. Esto permite validar la integridad del contexto del mensaje (por ejemplo, asegurarse de que proviene de un usuario específico o que tiene una fecha válida).

4 – Tenemos el siguiente jwt, cuya clave es “Con KeepCoding aprendemos”.

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzfQ.gfhw0dDxp6oixMLXXRP97W4TDTrv0y7B5YjD0U8ixrE

¿Qué algoritmo de firma hemos realizado?

**Algoritmo de firma utilizado:**

{

"typ": "JWT",

"alg": "HS256"

}

¿Cuál es el body del jwt?

El campo "alg": "HS256" muestra que se usó **HMAC con SHA-256** para firmar el token.

**Body del JWT:** El body (o payload) se encuentra entre el primer y segundo punto (.) del JWT, en formato Base64URL. Decodifiquémoslo:

eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzfQ

{

"usuario": "Don Pepito de los palotes",

"rol": "isNormal",

"iat": 1667933533

}

Un hacker está enviando a nuestro sistema el siguiente jwt:

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzQWRtaW4iLCJpYXQiOjE2Njc5MzM1MzN9.krgBkzCBQ5WZ8JnZHuRvmnAZdg4ZMeRNv2CIAODlHRI

¿Qué está intentando realizar?

**Hacker y su JWT manipulado:** El hacker envió este JWT:

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzQWRtaW4iLCJpYXQiOjE2Njc5MzM1MzN9.krgBkzCBQ5WZ8JnZHuRvmnAZdg4ZMeRNv2CIAODlHRI

Decodificando el body:

{

"usuario": "Don Pepito de los palotes",

"rol": "isAdmin",

"iat": 1667933533

}

**Intención del Hacker:**

* **Manipular el JWT** para cambiar el rol del usuario a isAdmin. Esto podría otorgarle privilegios administrativos en el sistema si la firma no fuera verificada.

¿Qué ocurre si intentamos validarlo con pyjwt?

**PyJWT verifica la firma del token**

Cuando llamas a jwt.decode(), **PyJWT**:

* Extrae el encabezado (header), payload y firma (signature) del JWT.
* Reconstruye la firma a partir del encabezado y el payload utilizando el algoritmo especificado (HS256 en este caso) y la clave secreta proporcionada.
* Compara la firma reconstruida con la firma que viene en el JWT.

**Resultado de la validación**

* **El hacker cambió el payload** del token (el rol pasó de isNormal a isAdmin), pero dejó la firma tal como estaba.
* Como el contenido del payload cambió, la firma reconstruida no coincidirá con la firma original incluida en el token.
* PyJWT detectará esto y lanzará un error de tipo InvalidSignatureError.

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?

**Hash SHA3 (Keccak):**

bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe

**Longitud:**

El hash tiene 64 caracteres hexadecimales. Dado que cada carácter hexadecimal representa 4 bits, esto equivale a 64×4=256 bits, correspondiendo a un **SHA3-256**.

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

4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833

**¿Qué hash hemos realizado?**

**Hash SHA2:**

4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833

**Longitud:**

El hash tiene 128 caracteres hexadecimales. Esto equivale a 128×4=512 bits correspondiendo a un **SHA-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?

**Efecto avalancha en acción**

El efecto avalancha se refiere a que un cambio mínimo en los datos de entrada produce un cambio completamente distinto en el resultado del hash. Este comportamiento garantiza que no haya correlación aparente entre los hashes de dos entradas similares.

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.

Se debe evidenciar la respuesta. Cuidado si se usan herramientas fuera de los lenguajes de programación, por las codificaciones es mejor trabajar en hexadecimal.

**857d5ab916789620f35bcfe6a1a5f4ce98200180cc8549e6ec83f408e8ca0550**

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?

SHA-1 (Secure Hash Algorithm 1) **no es seguro para proteger contraseñas** debido a varias razones:

* **Vulnerabilidad a colisiones:** SHA-1 está roto en términos de **resistencia a colisiones**. Investigadores han demostrado que pueden generar dos entradas diferentes que producen el mismo hash, lo cual compromete la integridad del algoritmo.
* **Fuerza bruta y ataques de diccionario:** SHA-1 es **rápido y eficiente**, lo cual es un problema para la seguridad de contraseñas, ya que permite realizar millones de intentos por segundo en ataques de fuerza bruta o con tablas precomputadas como **rainbow tables**.
* **Desactualización:** La mayoría de los estándares modernos (por ejemplo, NIST) ya no recomiendan el uso de SHA-1 desde hace años.

**Rainbow tables:** son tablas precomputadas que se utilizan para **romper contraseñas hasheadas** de manera eficiente. Son una forma optimizada de realizar **ataques de diccionario** contra contraseñas cifradas con algoritmos de hash.

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?

**MEJORAS:**

**Fortalecimiento del SHA-256: Hashing iterativo + Sal (Salt)**

Para **fortalecer SHA-256**, se puede implementar un esquema que combine **salting** e **iteraciones** (también conocido como **key stretching**):

* **Sal (salt):** Generar un valor aleatorio único (p. ej., 16 bytes) para cada contraseña y combinarlo con la contraseña antes de aplicar SHA-256. La salt evita ataques con rainbow tables.
* **Iteraciones (key stretching):** Aplicar SHA-256 **varias veces** en un bucle (miles de iteraciones) para aumentar el tiempo de cómputo requerido por cada hash. Esto hace que ataques de fuerza bruta sean significativamente más costosos.

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

**Propuesta de mejora: Algoritmos diseñados para contraseñas**

En lugar de usar SHA-256, que no fue diseñado específicamente para proteger contraseñas, se recomienda usar algoritmos **modernos y especializados**:

**a) PBKDF2 (Password-Based Key Derivation Function 2)**

* **Ventajas:** Es ampliamente soportado, configurable en iteraciones y seguro.
* **Uso:** Se basa en un algoritmo de hash como SHA-256 con múltiples iteraciones y sal.
* **Ejemplo práctico:** (ver código anterior).

**b) bcrypt**

* **Ventajas:**
  + Incorporación automática de una **salt**.
  + La cantidad de iteraciones (factor de costo) puede ajustarse dinámicamente.
  + Está diseñado para ser más lento y dificultar ataques por fuerza bruta.

**c) Argon2 (recomendado)**

* **Ventajas:**
  + Es el ganador del **Password Hashing Competition** (PHC).
  + Permite ajustar no solo el tiempo (iteraciones) sino también la memoria utilizada, lo que dificulta aún más los ataques con hardware especializado.

8 – Tenemos la siguiente API REST, muy simple.

Request:

Post /movimientos

Tabla

Descripción generada automáticamente

Petición de ejemplo que se desea enviar:

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

Response:

Tabla

Descripción generada automáticamente

Tabla, Calendario

Descripción generada automáticamente

{

"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?

**Redefinición de la API**

En la nueva versión de la API, los campos confidenciales deben cifrarse con **AES-GCM**, y se debe adjuntar un **HMAC-SHA-256** para garantizar la integridad. A continuación, un ejemplo de cómo se vería la petición y respuesta redefinida:

**Descripción de campos:**

1. idUsuario (no requiere confidencialidad).
2. datosCifrados:
   * Contiene los campos usuario y tarjeta cifrados con AES-GCM.
   * El resultado del cifrado (texto cifrado) se codifica en Base64 para su transmisión.
3. hmac:
   * Se calcula usando **HMAC-SHA-256** sobre datosCifrados.
   * Garantiza la integridad de los datos cifrados.

**Proceso Detallado**

1. **Cifrado de los campos confidenciales (AES-GCM):**
   * Se toma la información confidencial (usuario, tarjeta, movTarjeta, Saldo) y se cifra utilizando una **clave secreta simétrica** compartida entre el cliente y el servidor.
   * AES-GCM genera además un **tag de autenticación** que asegura que los datos no han sido alterados.
2. **Cálculo de la integridad (HMAC-SHA-256):**
   * Se calcula el HMAC usando una clave HMAC secreta y el contenido cifrado (datosCifrados).
   * El HMAC asegura que nadie puede modificar los datos sin ser detectado.
3. **Codificación en Base64:**
   * Tanto los resultados del cifrado como el HMAC se codifican en **Base64** para transmitirlos como JSON.
4. **Validación en el servidor:**
   * Al recibir la petición, el servidor:
     1. Calcula el HMAC del datosCifrados recibido y verifica si coincide con el HMAC adjunto.
     2. Si coincide, descifra datosCifrados utilizando la clave AES-GCM.
     3. Si alguno de estos pasos falla, se rechaza la petición.

**Beneficios de esta propuesta**

1. **Confidencialidad:** Los datos sensibles se cifran con AES-GCM.
2. **Integridad:** El uso de HMAC-SHA-256 garantiza que cualquier modificación no autorizada será detectada.
3. **Autenticación integrada:** AES-GCM incluye un **tag de autenticación** que protege los datos cifrados contra modificaciones.
4. **Eficiencia:**
   * El cifrado simétrico AES es rápido y eficiente.
   * HMAC es ligero y seguro.

**Consideraciones Adicionales**

* **Gestión de claves:**
  + Es crucial proteger las claves AES y HMAC. Se recomienda almacenarlas en un **HSM** (Hardware Security Module) o un gestor seguro de claves.
* **Rotación de claves:**
  + Se deben implementar mecanismos para la rotación periódica de claves.
* **Uso de sal:**
  + Para evitar patrones repetidos en los datos cifrados, puede añadirse una **sal aleatoria** en el cifrado AES-GCM.

**Alternativas de mejora**

Si se requiere aún mayor seguridad en el sistema end-to-end:

1. **Uso de cifrado asimétrico (RSA o ECC):**
   * En lugar de AES-GCM, los datos podrían cifrarse usando claves públicas del servidor, y solo el servidor con la clave privada podría descifrarlos.
2. **Implementar un protocolo completo como JOSE (JWT cifrado):**
   * Uso de **JWE (JSON Web Encryption)** con AES-GCM.
3. **TLS adicionalmente:**
   * Aunque el enunciado indica que no se usa TLS, si es posible en el futuro, se debe activar TLS para cifrado y autenticación a nivel de transporte.

Esta propuesta logra el equilibrio entre **seguridad** y **rendimiento**, mejorando significativamente la confidencialidad e integridad de la API.

9 – Se requiere calcular el KCV de las siguiente clave AES:

A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72

Para lo cual, vamos a requerir el KCV(SHA-256) así como el KCV(AES). El KCV(SHA-256) se corresponderá con los 3 primeros bytes del SHA-256. Mientras que el KCV(AES) se corresponderá con cifrar un texto del tamaño del bloque AES (16 bytes) compuesto con ceros binarios (00), así como un iv igualmente compuesto de ceros binarios. Obviamente, la clave usada será la que queremos obtener su valor de control.

**KCV(SHA-256):** DB7DF2

**KCV(AES):** 5244DB

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

KeepCoding© All rights reserved. www.keepcoding.io

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.

Texto

Descripción generada automáticamente

11 –

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:



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?

**Mensaje descifrado:** e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72

**Mensaje cifrado nuevamente:** 

Los textos cifrados son diferentes porque el algoritmo RSA-OAEP utiliza un enfoque probabilístico en su funcionamiento. Esto significa que cada vez que cifras el mismo mensaje con la misma clave pública, el texto cifrado será diferente. Esto sucede por las siguientes razones:

**Uso de un relleno aleatorio (random padding):**

* **OAEP (Optimal Asymmetric Encryption Padding)** introduce un elemento aleatorio como parte de su esquema de relleno.
* Este elemento aleatorio garantiza que, aunque cifres el mismo mensaje con la misma clave pública varias veces, el resultado será diferente cada vez.

12 – 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:E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74

Nonce:9Yccn/f5nJJhAt2S

¿Qué estamos haciendo mal?

**Reutilización del nonce:**

El nonce (número único) es fundamental para garantizar la seguridad de AES/GCM. Si reutilizas el mismo nonce con la misma clave en diferentes mensajes, introduces una vulnerabilidad grave. Esto puede permitir a un atacante realizar ataques de tipo:

* **Ataque de texto cifrado asociado (forged ciphertext):** El atacante puede manipular el texto cifrado y autenticado para descubrir el mensaje original.
* **Ataques de correlación:** Al tener dos mensajes cifrados con el mismo nonce, un atacante podría deducir patrones, lo que compromete la confidencialidad.

Cifra el siguiente texto:

He descubierto el error y no volveré a hacerlo mal

Usando para ello, la clave, y el nonce indicados. El texto cifrado presentalo en hexadecimal y en base64.

Mensaje JSON:

{

"nonce": "OVljY24vZjVuSkpo",

"tag": "ZNbU8OFVU+tg+XRD13YC4g==",

"texto\_cifrado": "efhnAcGwF61ZA5E7/g6jMGnCwwZKUKD2wCOxLq21/4nDV+smuCaRWW9BoMTbMp8UIi2X"

}

13 – Se desea calcular una firma con el algoritmo PKCS#1 v1.5 usando las claves contenidas en los ficheros clave-rsa-oaep-priv y clave-rsa-oaep-publ.pem del mensaje siguiente:

El equipo está preparado para seguir con el proceso, necesitaremos más recursos.

¿Cuál es el valor de la firma en hexadecimal?

**Firma RSA PKCS#1 v1.5 (hex):** 

Calcula la firma (en hexadecimal) con la curva elíptica ed25519, usando las claves ed25519-priv y ed25519-publ.

**Firma Generada (64 bytes):** b'42326a683949c9029dc0846ad214baed22c7bbce6e56fd65acefae463817ac8368ee892c08f55bc7e782ed6af1c3a8055c1564ba6b2ba3b6b3b75431f0b0c604'

14 – Necesitamos generar una nueva clave AES, usando para ello una HKDF (HMAC-based Extract-and-Expand key derivation function) con un hash SHA-512. La clave maestra requerida se encuentra en el keystore con la etiqueta “cifrado-sim-aes-256”. La clave obtenida dependerá de un identificador de dispositivo, en este caso tendrá el valor en hexadecimal:

e43bb4067cbcfab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3

Diagrama

Descripción generada automáticamente

¿Qué clave se ha obtenido?

**Salt utilizado (hex):** acb2779050ce8bb6e73607672672e1f5

**Clave AES derivada (hex):** f88d35679c37c04891223cbff37dc1cec0c1225928ea81f8f033542e28f714dc

15 – Nos envían un bloque TR31:

D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDBE6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E03CD857FD37018E111B

Donde la clave de transporte para desenvolver (unwrap) el bloque es:

A1A10101010101010101010101010102

**1. ¿Con qué algoritmo se ha protegido el bloque de clave?**

El bloque TR-31 utiliza **AES** como algoritmo para proteger el bloque de clave.

**2. ¿Para qué algoritmo se ha definido la clave?**

**Algoritmo 'A'**: Las operaciones de cifrado/descifrado de PINs utilizan el algoritmo **AES**.

**3. ¿Para qué modo de uso se ha generado?**

Modo de uso: B

**4. ¿Es exportable?**

El TR-31 no permite exportar claves fuera de un entorno seguro. Para determinar si la clave es exportable, deberíamos examinar más detalles del encabezado. En general, la exportación está controlada por políticas.

**5. ¿Para qué se puede usar la clave?**

El uso "B" en un bloque TR-31 indica que la clave contenida en el bloque está destinada para **operaciones de cifrado y descifrado de PINs** (números de identificación personal). Esto es típico en aplicaciones bancarias y de sistemas de puntos de venta, donde los PINs de las tarjetas deben protegerse durante su transmisión y almacenamiento.

**6. ¿Qué valor tiene la clave?**

Clave= c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1