Lab 1 - Ataque Padding Oracle

Autor: Felipe Torralba

Parte a)

Enunciado

Pruebe los servicios con distintos tipos de entrada (distintos largos, modificaciones de bytes,
etc). Documente el análisis exploratorio realizado y sus conclusiones.

Servidor A

Mismo mensaje

Al servicio A se le envía el mismo texto varias veces. El servidor responde con mensajes distintos.

Mensajes cortos

input -> `hola`
output -
> `4227bd9a0ef5fa5620355597f36364fd6ec4844c9f310ee3417049a7448b739e5c3a0e5b72dbc`
`fb282b3a677fbce3ba32dc344a673672a5e7e154587bd61aa095ba2dd8c8a2dd3abd2e61b30c95`
`0dcd2fb76f9340ecdd65ed11b05f7dc73ae0a13125bdeab9ede60e0e4c49a880c0688`
El largo en bytes de la respuesta es 112. El de el input fue 4 bytes.
Como el servidor usa CBC para encriptar, quiere decir que los primeros 16 bytes de la respuesta son el IV.
Restando eso, obtenemos que la cantidad de bytes de *padding* + mi texto encriptado + la contraseña son de largo 96 bytes.

input -> `hola!`
output -> ciphertext de largo en bytes 112
Al agregar un carácter, el largo del output en bytes siguen siendo 112 bytes.

input -> `a`
output -> ciphertext de largo en bytes 112

Incluso con un input de largo en bytes 1, la respuesta sigue siendo de 112 bytes.

Mensajes de largo variable

(largo se refiere a largo en bytes, no en hex)

input -> `hola1234`
output -> largo 112

input -> `hola12345`
output -> largo 128

Recién con un mensaje de largo 9, la respuesta del servidor es de largo 128.
Se le agregaron exactamente 16 bytes, que es el largo del input de un cifrador AES.

Servidor B

Modificando un byte arbitrario

Al servidor A le entrego `hola`, me devuelve un hex.
Ese hex lo convierto a un `bytearray`, luego modifico el byte en la posición 3 haciéndole un flip al bit menos significativo con XOR 1.
Al convertir ese `bytearray` a hex y entregándoselo al servidor B, éste responde "" (string vacío).

Modificando el último byte

Al servidor A le entrego `hola`, me devuelve un hex.
Ese hex lo convierto a un `bytearray`, luego modifico el último byte haciéndole un flip al su bit menos significativo.
Al convertir ese `bytearray` a hex y entregándoselo al servidor B, éste me responde

```
[Server] "pkcs7: invalid padding (last byte does not match padding)"
```

Parte b)

Enunciado

Cree un programa basado en el código base que envía un mensaje `m` al servidor A y le envía la respuesta del servidor A al servidor B. Documente lo observado.

Desarrollo

Se implementó un programa que realiza los siguientes pasos:

1. Crea conexiones al servidor A y al B
2. Le pide un input al usuario
3. Envía ese input al servidor A
4. Toma la respuesta del servidor A (en hex) y la envía al B
5. Muestra la respuesta de B

Observaciones

- Cuando se envía el ciphertext sin modificaciones desde A → B, el servidor B **responde correctamente** (no hay error).
- Si se modifica un **byte aleatorio intermedio** del ciphertext, el servidor B responde con mensajes de error del estilo

```
[Server B] "json: invalid character '\x17' in string literal"
```

- Si se modifica el **último byte** del ciphertext, el servidor B responde con:

```
[Server B] "pkcs7: invalid padding (last byte is larger than total
length)"
```

- Si se hace 0 el **último byte** del ciphertext, el servidor B responde con:

```
[Server B] "pkcs7: invalid padding (last byte does not match padding)"
```

Parte c)

Enunciado

¿Cómo podría conocer en un contexto genérico el tamaño del bloque del cifrador utilizado, sin conocer el cifrador? Solo responda teóricamente.

Respuesta

Una forma de conocer el tamaño del bloque del cifrador utilizado podría ser aumentando gradualmente el largo del input hasta observar cambios en el largo del texto cifrado.
Algo muy parecido se hizo en la experimentación de la parte a), donde el texto cifrado pasó de tener largo 112 a tener largo 128, lo cual dio una diferencia de 16, indicando el tamaño del bloque del cifrador.
La explicación de esto es que los input más cortos cabían en un bloque de 16 bytes, por lo cual el tamaño del texto cifrado no cambiaba. Al llegar a cierto largo, el modo de operación requerirá otro bloque, y como los cifradores vistos hasta el momento tienen un input de largo fijo, se puede obtener su tamaño.

Parte d)

Enunciado

Cree una función que permita descifrar el último carácter del texto cifrado.

Desarrollo

La idea es hacer una función que modifique el último byte del penúltimo bloque del texto cifrado, ya que el penúltimo bloque es el que se utiliza para obtener el último bloque del texto plano:

$$P_n = decrypt(C_n) \oplus C_{n-1}$$

Esto es lo que hace el servidor, para reconstruir cada bloque de P, cuando intenta desencriptar el mensaje C.
La única variable "controlable" de esta ecuación es C_{n-1}.
Entonces, con esta idea, para obtener el último byte del texto plano, se asume que debe existir por lo menos un $C'_{n-1}[15]$ (el [15] es por el último byte) que al hacerle XOR con $decrypt(C_n)[15]$ entregue 0x01 (es decir, un padding 0x01).
Como estamos iterando sobre todos los posibles valores de un byte, tenemos que probar todos los valores desde 1 hasta 255. Al tener el valor de $C'_{n-1}[15]$ que cumpla con lo anterior, se tiene lo siguiente:

$$0x01 = decrypt(C_n[15]) \oplus C'_{n-1}[15]$$

Despejando:

$$decrypt(C_n)[15] = 0x01 \oplus C'_{n-1}$$

Entonces, para recuperar el byte del texto plano original, volvemos a la primera ecuación:

$$P_n[15] = decrypt(C_n)[15] \oplus C_{n-1}[15]$$

Reemplazando:

$$P_n[15] = 0x01 \oplus C'_{n-1}[15] \oplus C_{n-1}[15]$$

Con esta idea en mente (en el código):

1. Se separa el texto cifrado en bloques de 16 bytes
2. Se guarda el penúltimo bloque
3. Para encontrar un byte de C'_{n-1} que cumpla con lo explicado anteriormente, se itera sobre todos sus posibles 256 valores. En cada iteración:
   1. Se verifica que el valor que estamos probando `guess` no sea igual al byte del mensaje original $C_{n-1}[15]$. Si lo es, se ignora y se pasa al siguiente `guess`
   2. Se crea una copia usando `deepcopy` (debido a que en Python los arreglos son mutables) de los bloques cifrados originales
   3. Se modifica el último byte de C'_{n-1} con nuestro *guess*
   4. Se arma un mensaje C' para enviar al servidor B
   5. Se verifica si la respuesta del servidor no contiene `invalid padding` ya que eso indica que el padding fue válido (0x01, ya que solo se modificó un byte de C).
   1. Si el padding fue válido, se encontró un $C'_{n-1}[15]$ que sirve para la ecuación. Con este valor, se calcula $P_n[15]$ y se retorna.
   2. Si no fue válido, se continúa con el siguiente *guess*, repitiendo el proceso desde el paso 3

   Si no se encuentra un valor válido para $C'_{n-1}[15]$, el programa continúa su ejecución dejando ese byte en 0.

Parte e)

Enunciado

Modifique la función anterior para descifrar un bloque completo.

Desarrollo

Para descifrar los siguientes caracteres, hay que *fijar* los k bytes (con k entre 15 e i, el actual byte que se quiere descifrar) de C'_{n-1} anteriores. Esto se hace calculando

$$C'_{n-1} = decrypt(C_n)[k] \oplus pad$$

Siendo pad el padding para esa iteración (si se calcula el byte 13, pad debería ser 0x03).
Luego, para calcular el valor del componente que falta

$$decrypt(C_n)[k] = P_n[k] \oplus C_{n-1}[k]$$

Reemplazando

$$C'_{n-1} = P_n[k] \oplus C_{n-1}[k] \oplus pad$$

Esto asume que los k bytes anteriores de P_n ya fueron descifrados, por lo que no aplica para la iteración con i=15 (la primera que se ejecuta).
Luego, se descifra el byte $P_n[i]$ usando el mismo procedimiento que en la parte d.
Esto se repite para todos los bytes del bloque, desde la posición 15 hasta 0.

Resultados

A continuación, se muestra el resultado del ataque para distintos inputs.
input -> "hola"
output ->

```
Deciphered message:
{
"name":"hola",
"secret":"83c804bf74f04b841b9c186342c62a1797cd87b1fe6ce7bf54d32c4a16390020
"
}
```

input -> "hola que tal"
output->

```
Deciphered message:
{
"name":"hola que tal",
"secret":"83c804bf74f04b841b9c186342c62a1797cd87b1fe6ce7bf54d32c4a16390020
"
}
```

input->"this is a encrypted message"
output->

```
Deciphered message:
{
"name":"this is a encrypted message",
"secret":"83c804bf74f04b841b9c186342c62a1797cd87b1fe6ce7bf54d32c4a16390020
"
}
```

A partir de estos resultados, se puede ver que la llave *key* especificada en *secret* es:

```
83c804bf74f04b841b9c186342c62a1797cd87b1fe6ce7bf54d32c4a16390020
```

Discusión

Si no se encuentra un $C'_{n-1}[i]$ que cumpla con lo requerido y no sea el byte del mensaje original $C_{n-1}[i]$, anteriormente, las funciones tenían un *fallback* para que se elija $C_{n-1}[15]$. Esto tiene sentido ya que en realidad el servidor B sí respondió que el padding era válido, pero al usarlo para calcular $P_n[i]$ se tiene:

$$P_n[i] = pad \oplus C_{n-1}[i] \oplus C_{n-1}[i]$$

Lo cual, por cómo funciona $\oplus$, siempre resulta en el valor que tenga el *padding*.
No se considera un byte obtenido de este modo como válido, ya que se hicieron un par de pruebas y al parecer que no encuentre un $C'_{n-1}[i]$ entre los 256 posibles valores solo ocurre ocasionalmente.
Por ejemplo, para el input `hola`, primero se obtuvo

```
{
"name":"hola",
"secret":"83c804bf74f04b841b9c186342c62a1797cd87b\x10\x0f\x0e\r\x0c\x0b\n\
t\x08\x07\x06\x05\x04\x03\x024a16390020"
}
```

Aquí, hubo parte de un bloque que se reemplazó con los valores del *padding* usados para calcular ese byte, en particular fue del padding 10 al 2

```
\x10\x0f\x0e\r\x0c\x0b\n\t\x08\x07\x06\x05\x04\x03\x02
```

Finalmente, se optó por desechar este *fallback* ya que no ofrecía mayor valor que simplemente dejar los bytes incorrectos en 0.