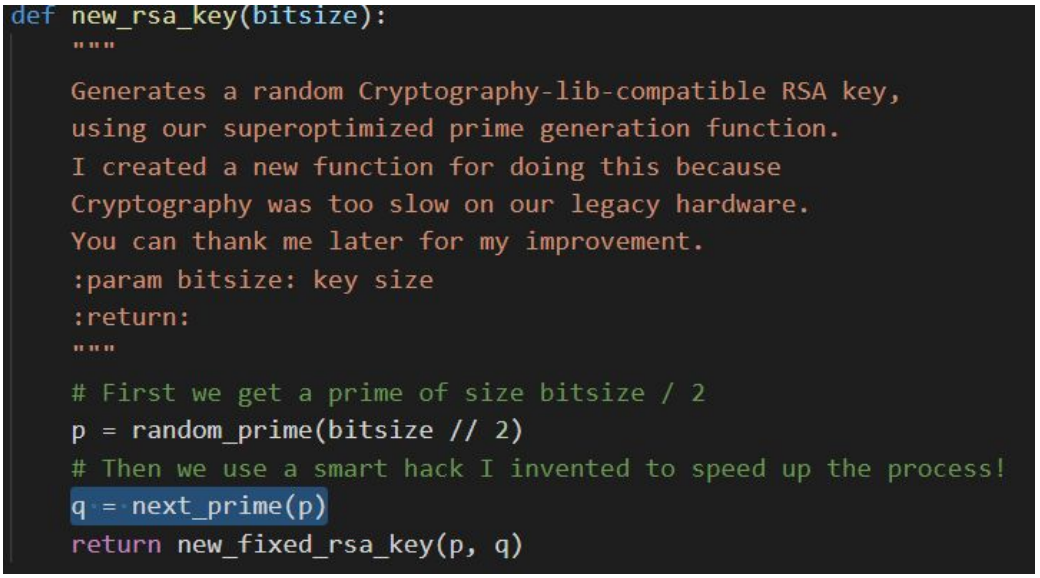
Laboratorio 1:

Quebrando RSA:

1. Dentro de la implementación de RSA propuesta se rompe uno de los principios importantes que dificultan a un adversario obtener la factorización del número N.



```python
def new_rsa_key(bitsize):
    """
    Generates a random Cryptography-lib-compatible RSA key,
    using our superoptimized prime generation function.
    I created a new function for doing this because
    Cryptography was too slow on our legacy hardware.
    You can thank me later for my improvement.
    :param bitsize: key size
    :return:
    """
    # First we get a prime of size bitsize / 2
    p = random_prime(bitsize // 2)
    # Then we use a smart hack I invented to speed up the process!
    q = next_prime(p)
    return new_fixed_rsa_key(p, q)
```

Particularmente en el archivo utils.py se encuentra la función new_rsa_key, el que obtiene, a partir de p y q generados, la llave RSA (la llave privada y llave pública).

El problema es que ambos valores (p y q) no son primos aleatorios, sólo p lo es y q corresponde al número primo siguiente de p, como se puede apreciar en la línea de código marcada (q = next_prime(p)), en otras palabras se rompe la condición de que ambos números sean aleatorios.

Lo que debería hacerse para cumplir con la seguridad que ofrece RSA es hacer que tanto p y q sean números primos aleatorios, por lo que, para corregir la vulnerabilidad, la línea marcada debería ser q = random_prime(bitsize//2), lo que asegura lo enunciado.

2. Funcionamiento de RSA:

El algoritmo de generación de claves de RSA funciona a partir de los siguientes valores p,q primos (aleatorios), que definen N:=p*q y phi(N):=(p-1)*(q-1), un valor aleatorio e (generalmente 3 o 65537) que cumple con la propiedad matemática que mcd(e, phi(N) = 1 y d, que corresponde a un valor acotado entre 1 y phi(N).

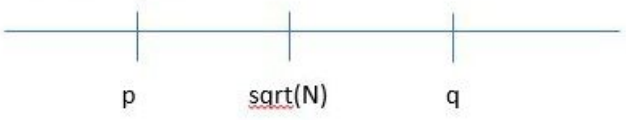
A partir de estos valores se genera una clave pública pk=(e,N) y una clave privada sk =(d,p,q). Las operaciones que realiza RSA sobre texto plano corresponde a la encriptación en la que se utilizan los valores de la clave pública (e y N) en conjunto con el texto plano (M) y el desencriptado que utiliza los valores de la clave privada (d y N) en conjunto con el mensaje cifrado (C).

De acuerdo a este funcionamiento, para poder descifrar un mensaje es necesario conocer la clave privada (d y N), sin embargo para obtener estos valores es necesario obtener la factorización de un número, la que es computacionalmente costosa o “difícil”.

Abusando de la vulnerabilidad:

Sin embargo, dadas las condiciones del sistema de generación de claves del sistema estudiado, tenemos el valor de e y si tenemos p tenemos q o viceversa, debido a la vulnerabilidad presentada en el apartado anterior, pues q es el número primo siguiente a p.

Debido a este problema se puede observar lo siguiente:




| P | Q | N | sqrt(N) |
|---|---|---|---|
| 2 | 3 | 6 | 3.46 |
| 3 | 5 | 15 | 3.87 |
| 5 | 7 | 35 | 5.91 |
| 7 | 11 | 77 | 8.77 |
| 11 | 13 | 143 | 11.95 |
| 13 | 17 | 221 | 14.86 |

Esto es un problema pues si se encuentra uno de los dos números, es posible obtener uno a partir de otro, reduciendo drásticamente la búsqueda y, por la implementación de RSA, a partir de los valores de p y q, en conjunto con e (que es fijo en este caso), es posible obtener tanto la llave pública como la llave privada, por

lo que permitiría que un adversario pueda desencriptar cualquier mensaje encriptado con la llave pública.

Veamos que esto se cumple para todo primo p y q consecutivos:

Es sabido que (pues q es el primo siguiente de p):

$$p * q = N \wedge p < q$$

Luego:

$$p < q \Rightarrow p * p < p * q = N \Rightarrow p < \sqrt{N}$$

$$p < q \Rightarrow N = p * q < q * q \Rightarrow \sqrt{N} < q$$

Por lo tanto:

$$p < \sqrt{N} < q,\ p \approx \sqrt{N} \wedge q \approx \sqrt{N}$$

De esta forma podemos obtener la factorización prima de N (que es única por el teorema fundamental de la aritmética), a partir de obtener el primo más cercano hacia adelante (q) y hacia atrás (p) de sqrt(N), o utilizando el teorema, deducir uno de los dos valores utilizando N/p = q o N/q = p de acuerdo al valor que obtengamos.

Ilustrativamente se realizaría lo siguiente:

$$q = next\_prime(\sqrt{N})$$
$$p = N/q$$

A partir de los valores de p y q obtenidos es posible obtener phi(N), a partir del que es posible obtener d pues e es dato, se realiza e*d MOD phi(N) = 1 (obtener inverso multiplicativo de e), a partir del que, finalmente, se puede obtener el mensaje desencriptado pues se tienen los datos de la llave privada.

Ilustrativamente se realizaría lo siguiente:

$$\Phi(N) = (p-1) * (q-1)$$
$$\left\{ d \in \mathbb{Z}^+ \mid ed\ mod\ \Phi(N) = 1 \right\}$$
$$sk = (d, p, q)$$

¿Cómo ejecutar el script? (README):

1. Posicionarse en la carpeta correspondiente a la P1
2. Ejecutar main.py para obtener los archivos encriptados
3. Ejecutar findprimes.py para obtener el texto plano

Padding Oracle:

1. En el servicio A, al variar el largo de la entrada, varía el largo del mensaje encriptado recibido.
   En el servicio B, al variar un carácter con respecto a la encriptación original se recibe un error de padding inválido:
   ```
   [Server] "pkcs7: invalid padding (last byte is larger than total length)"
   ```
   En el servicio B, al agregar un carácter que no corresponde a hexadecimal, por ejemplo q, se recibe un error de codificación:
   ```
   [Server] "hex: encoding/hex: invalid byte: U+0071 'q'"
   ```

2. Código realizado
3. La estrategia consiste en enviar un mensaje que aumenta su tamaño en un byte y verificar el largo del mensaje encriptado recibido, cuando el largo cambia, verificar cada cuántos bytes cambia ese valor, probar esto hasta que el mensaje enviado sea distinto a la desencriptación del mensaje al enviarlo al servidor B (utilizando la función implementada en el apartado anterior). Finalmente el tamaño de bloque corresponde a la tendencia encontrada, que fue almacenada en una lista.

¿Cómo ejecutar el script? (README):

1. Posicionarse en la carpeta correspondiente a la P2
2. Ejecutar main.py para realizar ataque

```
Vamos en el byte: 1
Vamos en el byte: 0
bytearray(b'{"name":"hola","')

bytearray(b'{"name":"hola","')
bytearray(b'secret":"359a68f')
bytearray(b'8be154aceaa4d0d7')
bytearray(b'617a76dfd390da1c')
bytearray(b'762ff51eea0ba48c')
bytearray(b'c8a1d81bc"}\x05\x05\x05\x05\x05')

[Server] Ciphertext: "dff3289ba309388282c43ca9b937440e1b26337d567f252645fc850ab478122635dc49df2f46d2004
4d332d83c8cd0f6f056385b0d41d13cb2e8fb9fb9e224398e73646ced2f0969e032b70828f2c998dabb5e9053e2bb5205d23afd
0f1722843853d71c6294db97ec7f80282f210ef8"
send a message:
```

Integrantes:

- Maximiliano Vargas
- Pablo F. Torres Gutiérrez