4 - Random numbers

April 2, 2023

1 Tema 3 : Creación de azar (RNG)

Los humanos no somos buenos creando azar. Caemos en muchas trampas:

- "¡El número 12345 no pueder ser aleatorio!"
- "Lleva 20 años sin que la lotería acabe en 5, este año tiene que salir"

Los ordenadores tampoco son mucho mejores para crear azar. Son máquinas deterministas que siempre hacen lo mismo de acuerdo con una receta, así que pueden similar azar pero no crearlo realmente.

En criptografía, necesitamos una fuente de azar en varios momentos:

- Cuando creamos una clave simétrica, tiene que ser totalmente aleatoria
- Cuando creamos nonces o IV, tienen que ser totalmente aleatorios
- Cuando creamos números aleatorios para RSA, tienen que ser realmente aleatorios

Si cualquiera de estos números no fuesen perfectamente aleatorios estamos comprometiendo la seguridad del sistema porque un atacante podría adivinar algunos de esos números.

1.1 Fuentes de azar no criptográficas

Veremos algunas fuentes de azar comunes que no son criptográficas, y por tanto no debemos utilizarla en nuestros algoritmos.

En Python, el paquete `random` se encarga de simular números al azar. **CUIDADO: el paquete random no es una fuente de azar válida para algoritmos criptográficos**

[1]:
```
import random
```

[2]:
```
print(random.random())
print(random.random())
print(random.random())
```

```
0.7765644953276147
0.6034516832981685
0.44824484158009126
```

[3]:
```
print(random.random())
print(random.random())
print(random.random())
```

```
0.5346812681069882
0.4680284227253152
0.20584592598282114
```

En criptografía normalmente necesitamos un conjunto de bytes aleatorios, no un número. Podríamos convertir series de números reales a series de bytes así, **pero no lo hagas**.

[4]:
```
''.join(chr(int(255 * random.random())) for _ in range(0, 5)).encode()
```

[4]:
```
b'nW\xc3\x94\xc2\xa2\xc2\xbb'
```

Si ponemos una semilla, podemos repetir los números "al azar". Esto es útil, por ejemplo, en juegos, para que todos tengan las mismas cartas repartidas al azar

[5]:
```
random.seed(0.12345)
print(random.random())
print(random.random())
print(random.random())
```

```
0.694497180042137
0.6795321732387855
0.3290009188527938
```

[6]:
```
random.seed(0.12345)
print(random.random())
print(random.random())
print(random.random())
```

```
0.694497180042137
0.6795321732387855
0.3290009188527938
```

No sé cómo está programada la función `random.random()` de Python. Una manera común es utilizar la zona caótica de la aplicación logística, como el ejemplo que viene a continuación.

Fíjate que siempre necesita una semilla: si no le das una, toma el número de microsegundos de este momento según `time.time()`

[7]:
```
def rand_source(x=None):
    """ Generación de número al azar usando aplicación logística.

    Semilla: cualquier número real entre 0 y 1.

    Si no hay semilla, usa la parte decimal de time.time()
    """
    if x == None:
        import time
        _, x = divmod(time.time(), 1)
    while True:
        x = 4 * x * (1 - x)
```

```
        yield x
```

[29]:
```
rand = rand_source()
print(next(rand))
print(next(rand))
print(next(rand))
```

```
0.9353295100727337
0.24195287063933466
0.7336467161148802
```

Si fijamos la semilla, podemos obtener siempre los mismos números aleatorios

[9]:
```
rand = rand_source(0.12345678)
print(next(rand))
print(next(rand))
print(next(rand))
rand = rand_source(0.12345678)
print(next(rand))
print(next(rand))
print(next(rand))
```

```
0.43286081388812636
0.9819693187529408
0.07082230312330459
0.43286081388812636
0.9819693187529408
0.07082230312330459
```

La ventaja de generar números aparentemente aleatorios con la aplicación logística es que es rapidísimo y muy sencillo de programar. Las desventajas son que se puede adivinar más o menos dónde caerá el próximo número dado el anterior. Ejecutad varias veces el ejemplo sin semilla: veréis que después de un número mayor de 0.9 es muy probable que paséis a un número menor de 0.3.

Además, a veces es posible conocer la semilla inicial: porque un atacante pregunta la hora justo a la vez que se crea el generador de números aleatorios, por ejemplo.

Esta forma, o alguna similar, es la manera habitual de crear números aleatorios en las librerías de programación. Pero **no es segura desde un punto de vista criptográfico**. `random.random()` o similares no se deben utilizar cuando necesitamos números aleatorios en nuestros sistemas seguros.

1.2 Acceso a generadores seguros del sistema operativo

Casi todos los sistemas operativos permiten acceder a generadores seguros de secuencias aleatorias con algún método hardware no algorítmico.

En Linux estos generadores utilizan `/dev/urandom`, que es (¡más o menos!) el cifrado ChaCha20 del ruido que viene del teclado. Windows utiliza otros mecanismos similares.

Esta función *urandom* del sistema operativo, o las que ofrece el módulo Crypto, sí que se pueden utilizar en algoritmos criptográficos. Por ejemplo, se pueden utilizar para generar la clave de

algoritmos de cifrado simétrico, o sus *nonce*, o sus *IV*

[32]:
```
import os
print(os.urandom(5))
```

```
b'\xa0t\x1ew\xdd'
```

[34]:
```
!python3 -m pip install pycryptodome

import Crypto.Random
Crypto.Random.get_random_bytes(5)
```

```
Requirement already satisfied: pycryptodome in
/home/gattes/.cache/pypoetry/virtualenvs/jupyter-notebooks-QRvsWska-
py3.10/lib/python3.10/site-packages (3.17)

[notice] A new release of pip is
available: 23.0 -> 23.0.1
[notice] To update, run:
pip install --upgrade pip
```

[34]:
```
b'pFL\xed3'
```

Fíjate que a estas funciones no les puedes pasar una semilla, porque están basadas en generadores de ruido real.

1.3 Fuente de aletoriedad criptográfica con semilla

Una manera típica de implementar un PRNG es utilizar la fase PRNG de un cifrado de flujo. La clave es la semilla y el frujo de datos a cifrar son ceros. Por ejemplo, con ChaCha20:

[35]:
```
from Crypto.Cipher import ChaCha20
from Crypto.Random import get_random_bytes

def rand_source(x=None):
    """ Generación de número 'al azar' pero replicable usando ChaCha20.

    Si no hay semilla, usa una clave al azar
    """
    if x == None:
        x = get_random_bytes(32)
    cipher = ChaCha20.new(key=x, nonce=None)
    while True:
        yield cipher.encrypt(chr(0).encode())
```

[41]:
```
rand = rand_source()
print(next(rand))
print(next(rand))
print(next(rand))
```

```
b'\xf3'
b'\xbe'
b'\xb0'
```