

# Breaking Random( ) Internals For Fun!

Ismael Gómez Esquilichi

# # whoami



Estudiante de Ingeniería de la Ciberseguridad

Becario en el grupo de investigación GRAFO

Seguridad ofensiva en Vexcel Spain

A veces juego CTFs con *base64* y *John Keys*

Intentando romper cacharros con *IoTaK0s*

@esquilichii en  y 



# # Índice

1. Introducción
2. `java.util.Random()`
3. Generador Congruente Lineal (LCG)
4. Random String Utils (RSU)
5. Estado del arte de los distintos ataques
6. `SecureRandom()` como alternativa
7. Conclusiones

# # Introducción

- Es muy importante elegir números aleatorios en un montón de aplicaciones
- No se le presta mucha atención, pero es muy importante poder generar números aleatorios en un ordenador



# # Introducción

- Es muy importante elegir números aleatorios en un montón de aplicaciones
- No se le presta mucha atención, pero es muy importante poder generar números aleatorios en un ordenador
- Si nos llevamos este concepto a una persona... ¿Somos buenos generadores de números?
  - No, si pedimos a mucha gente elegir un número entre 0 y 100 observamos lo siguiente
    - Son mucho más frecuentes los acabados en 3 y 7
    - Muy poco frecuentes los acabados en 0 y 5
- ¿Lo harán mejor las máquinas?



# # Introducción

- Entonces, ¿que hacen las funciones que generan números aleatorios en distintos lenguajes?
  - Python -> `random()`
  - PHP -> `rand()` & `mt_rand()`
  - C -> `rand()`
  - Java -> `Random()`
- Cada implementación es distinta, pero todas las mencionadas generan números *pseudoaleatorios*, ya que para una máquina es imposible generar números aleatorios puros
- La mayoría de ellas se enfocan en conseguir números suficientemente aleatorios para diversos casos de uso

# # Introducción

GitHub Advisory Database / Unreviewed / CVE-2022-40267

## Predictable Seed in Pseudo-Random Number Generator (PRNG)...

**Critical severity** Unreviewed Published on Jan 20 to the GitHub Advisory Database • Updated on Apr 26

### Package

No package listed— Suggest a package

### Affected versions

Unknown

### Patched versions

Unknown

### Description

Predictable Seed in Pseudo-Random Number Generator (PRNG) vulnerability in Mitsubishi Electric Corporation MELSEC iQ-F Series FX5U-xMy/z (x=32,64,80, y=T,R, z=ES,DS,ESS,DSS) with serial number 17X\*\*\*\* or later, and versions 1.280 and prior, Mitsubishi Electric Corporation MELSEC iQ-F Series FX5U-xMy/z (x=32,64,80, y=T,R, z=ES,DS,ESS,DSS) with serial number 179\*\*\*\* and prior, and versions 1.074 and prior, Mitsubishi Electric Corporation MELSEC iQ-F Series FX5UC-xMy/z (x=32,64,96, y=T, z=D,DSS)) with serial number 17X\*\*\*\* or later, and versions 1.280 and prior, Mitsubishi Electric Corporation MELSEC iQ-F Series FX5UC-xMy/z (x=32,64,96, y=T, z=D,DSS)) with serial number 179\*\*\*\* and prior, and versions 1.074 and prior, Mitsubishi Electric Corporation MELSEC iQ-F Series FX5UC-32MT/DS-TS versions 1.280 and prior, Mitsubishi Electric Corporation MELSEC iQ-F Series FX5UC-32MT/DSS-TS versions 1.280 and prior, Mitsubishi Electric Corporation MELSEC iQ-F Series FX5UC-32MR/DS-TS versions 1.280 and prior, Mitsubishi Electric Corporation MELSEC iQ-R Series R00/01/02CPU all versions, Mitsubishi Electric Corporation MELSEC iQ-R Series R04/08/16/32/120(EN)CPU all versions allows a remote unauthenticated attacker to access the Web server function by guessing the random numbers used for authentication from several used random numbers.

### References

- <https://nvd.nist.gov/vuln/detail/CVE-2022-40267>
- <https://jvn.jp/vu/JVNVU99673580/index.html>
- <https://www.cisa.gov/uscert/ics/advisories/icsa-23-017-02>
- [https://www.mitsubishielectric.com/en/psirt/vulnerability/pdf/2022-019\\_en.pdf](https://www.mitsubishielectric.com/en/psirt/vulnerability/pdf/2022-019_en.pdf)

### Severity

**Critical** 9.1 / 10

### CVSS base metrics

Attack vector	Network
Attack complexity	Low
Privileges required	None
User interaction	None
Scope	Unchanged
Confidentiality	High
Integrity	High
Availability	None

CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N

### Weaknesses

CWE-335 CWE-337

### CVE ID

CVE-2022-40267

### GHSA ID

GHSA-9vv4-3cf7-mqqr

# # java.util.Random( )

- Es una clase en Java que se utiliza para generar secuencias de números pseudoaleatorios
  - Aunque los números que genera parecen aleatorios, en realidad son producidos de una manera determinista basada en una semilla inicial
- Tiene usos muy variados y distintos
  - Generación de Tokens (hmmm...)
  - Simulaciones
  - Generación de datos
  - Testing
  - Juegos



# # java.util.Random( )

- Está implementado como un generador congruente lineal (LCG)
- Funciona muy rápido ya que solo se compone de 4 operaciones
  - {MUL - SUM - AND - ROL}
- No es criptográficamente seguro, se puede predecir el estado
- Por defecto, usa *System.nanoTime()* como semilla si no se especifica una

# # java.util.Random( )

```
[ubuntu@armvps ~/random_internals/demo1]$
```

# # java.util.Random( )



```
private static final long multiplier = 0x5DEECE66DL;
private static final long addend = 0xBL;
private static final long mask = (1L << 48) - 1;

protected int next(int bits) {
    long oldseed, nextseed;
    AtomicLong seed = this.seed;
    do {
        oldseed = seed.get();
        nextseed = (oldseed * multiplier + addend) & mask;
    } while (!seed.compareAndSet(oldseed, nextseed));
    return (int)(nextseed >>> (48 - bits));
}
```

# # java.util.Random( )

- Vamos a hablar del método *nextInt(int bound)*
- Genera un número aleatorio entre 0 (inclusivo) y el parámetro *bound* (exclusivo)
- Se hace una llamada a *next(31)* para generar un entero de 32 bits (ignorando el bit de signo evitando negativos)
- Si *bound* es potencia de 2, se puede obtener el número multiplicando
  - $(r * bound) \gg 31$
- Si no se cumple, se hace un bucle para garantizar uniformidad
  - Poco probable cuando *bound* es pequeño, para romper RSU *nos da igual*

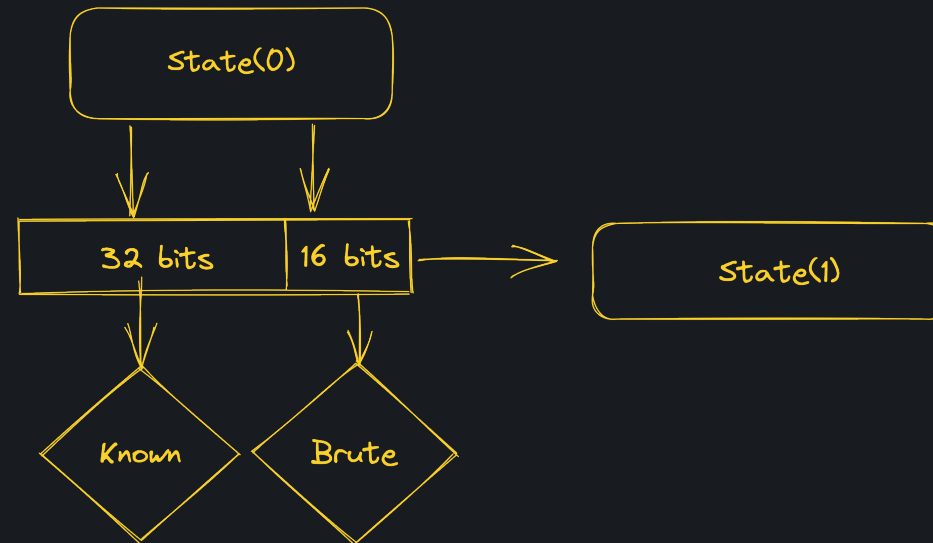
## # Generador Congruente Lineal

$$number_i + 1 = (A * number_i + C) \bmod M$$

$$A = 0x5DEECE66D, C = 0xB, M = 2^{48}$$

# # Generador Congruente Lineal

- Si conocemos el valor de  $x_i$  podemos predecir  $x_{i+n}$
- No conocemos todo el valor de  $x_i$  ya que Random() no da más de 32 bits del estado interno (semilla)
  - Ataque de fuerza bruta los 16 bits restantes es trivial, milisegundos



# # Generador Congruente Lineal

```
~/src/random_internals/demo2
```

# # Random String Utils

- “*Apache Commons Lang 3*” es una librería que proporciona todo tipo de utilidades
  - Siendo una de ellas la clase `RandomStringUtils`
- La llamada al método `RandomStringUtils.randomAlphanumeric(count)` devuelve un string que contiene *count* caracteres
- En la propia documentación de la clase, está escrito que la implementación NO es criptográficamente segura
  - Únicamente se contempla su uso en casos muy simples

org.apache.commons.lang3

## Class RandomStringUtils

java.lang.Object

org.apache.commons.lang3.RandomStringUtils

```
public class RandomStringUtils  
extends Object
```

Generates random Strings.

**Caveat:** Instances of **Random**, upon which the implementation of this class relies, are not cryptographically secure.



# # Random String Utils



```
def random_alphanumeric(count):  
    start = ord(' ')  
    end = ord('z') + 1  
    gap = end - start # gap == 91  
    out = ''  
    while len(out) < count:  
        code_point = next_int(gap) + start  
        if chr(code_point).isalnum():  
            out += chr(code_point)  
    return out
```

# # Random String Utils

- Para romper RSU, el problema en realidad es romper *Random.nextInt(91)* dado outputs *casi* consecutivos
  - Se computa *nextInt(91) + 32*
  - Hay 62 caracteres
  - El  $62/91 \approx 68.1\%$  de las veces la letra generada será usada
  - El  $29/91 \approx 32.9\%$  de las veces será una letra no válida y habrá que generar otra
  - Como consecuencia, si la letra no es válida, nos saltamos un output de *nextInt(91)*

# # Random String Utils

```
~/sr/r/demo3/random-internals
```

# # Random String Utils (1º Ataque)

- $y_i = c_i - 32$  será el output  $i$  de `nextInt(91)`
  - Asumiremos que no nos saltamos ninguna generación

# # Random String Utils (1º Ataque)

- $y_i = c_i - 32$  será el output  $i$  de *nextInt(91)*
  - Asumiremos que no nos saltamos ninguna generación

$$y_i = (x_i \gg 17) \bmod 91$$

- $x_i$  es el estado del LCG (48 bits).
- La operación ROR (Rotate Right) se hace por *next(31)* que se llama desde *nextInt(91)*

# # Random String Utils (1º Ataque)

- $y_i = c_i - 32$  será el output  $i$  de *nextInt(91)*
  - Asumiremos que no nos saltamos ninguna generación

$$y_i = (x_i \gg 17) \bmod 91$$

- $x_i$  es el estado del LCG (48 bits).
- La operación ROR (Rotate Right) se hace por *next(31)* que se llama desde *nextInt(91)*
- Podemos reescribir esta ecuación como la siguiente:

$$y_i = (x_i \gg 17) + 91k_1$$

## # Random String Utils (1º Ataque)

$$y_i = (x_i \gg 17) + 91k_1$$

- $k_1$  es un entero que satisface la restricción  $|k_1| < [2^{31}/91]$ 
  - Recordemos que  $x_i$  es un número de 31 bits
  - Eso significa que  $k_1$  debe ser tal que cuando 91 se multiplique por él, no supere  $2^{31}$

## # Ejemplo

$$y_i = (x_i \gg 17) + 91k_1$$

Para  $x_i = 125829120$

1.  $x_i \gg 17 = 960$
2.  $y_i = 960 \bmod 91 = 50$
3. Si  $y_i = 50$ , la ecuación queda como  $50 = 960 + 91k_1$
4. Por lo tanto,  $k_1 = -10$



# # Random String Utils (1º Ataque)

- Esta ecuación nos da la idea para el primer ataque mejor que hacer fuerza bruta a  $2^{48}$  posibles valores de semilla

$$y_i = (x_i \gg 17) + 91k_1$$

1. Brute force a todos los posibles valores de  $k_1$
  2. Calcular  $y_i - 91k_1$  nos da un candidato para  $x_i \gg 17$
  3. Para cada candidato, realizar fuerza bruta sobre todos los  $2^{17}$  posibles valores
  4. La cantidad total de fuerza bruta requerida para este ataque es de aproximadamente  $\left\lceil \frac{2^{31}}{91} \right\rceil * 2^{17} \approx 2^{41.5}$
- Técnica descubierta por [alex91ar/randomstringutils](https://github.com/alex91ar/randomstringutils)

# # Random String Utils (1º Ataque)

```
[ubuntu@armvps ~/src]$ ./random -b
Cracks RandomStringUtils.RandomAlphaNumeric()
By Alejo Popovici @alex91dotar (Apok)

=== Benchmark Mode ===
[*] Original token provided: TPIjPBmthTbDqZVH3G0XUW6rJoy7rV
[*] Size of token:30
[+] Cracking state...
[*] Number of threads: 4
[*] Starting from state: 211039265751040.
[*] State = 212602633940717
[*] 40.5 bits cracked in 31.530331 seconds.
[*] To crack the state would take a maximum of 5707.599609 seconds.
[*] Or 1.585444 hours.
[+] Bye!
```

## # Random String Utils (2º Ataque)

- Vamos a seguir mirando esta ecuación a ver si conseguimos sacar algo más de información que solo usando un único output

$$y_2 = (x_2 >> 17) \bmod 91$$

- Se puede escribir la anterior ecuación como:

$$y_2 = (((Ax_1 + C) \bmod M) >> 17) \bmod 91$$

- Lo único que no conocemos en esta ecuación es  $x_i$

## # Random String Utils (2º Ataque)

$$x_1 = x_{1,U_{31}} + x_{1,L_{17}}$$

$$y_2 = (((A(x_{1,U_{31}} + x_{1,L_{17}}) + C) \bmod M) \gg 17) \bmod 91$$

$$\implies y_2 = (((Ax_{1,U_{31}} + Ax_{1,L_{17}} + C) \bmod M) \gg 17) \bmod 91$$

$$\implies y_2 = (((((Ax_{1,U_{31}} + C) \bmod M) + (Ax_{1,L_{17}} \bmod M)) \bmod M) \gg 17) \bmod 91$$

- Hemos separado  $Ax_{1,U_{31}} + C$  y  $Ax_{1,L_{17}}$

## # Random String Utils (2º Ataque)

- Los valores que hemos separado en la anterior diapositiva como máximo pueden ser  $M - 1$
- Con esa premisa, tras sumar cada parte y hacer su módulo, al hacer otro módulo como mucho restaremos  $M$  o no se hará nada
- Aprovechándonos de eso, podemos reescribir la ecuación así:

$$y_2 = (((((Ax_{1,U_{31}} + C) \bmod M) \\ + (Ax_{1,L_{17}} \bmod M) - b_2M) \gg 17) \bmod 91$$

- Donde  $b_2 \in \{0, 1\}$

## # Random String Utils (2º Ataque)

- Ahora queremos separar  $Ax_{1,U_{31}} + C$  y  $Ax_{1,L_{17}}$  a través de la operación de desplazamiento a la derecha
- $c$  es un término de corrección

$$(a + b) \gg n = (a \gg n) + (b \gg n) + c, \quad c \in \{0, 1\}$$

- Si tenemos en consideración que  $M = 2^{48}$  y tenemos  $b_2 2^{31} + c_2$

$$y_2 = (((Ax_{1,U_{31}} + C) \bmod M) \gg 17) \\ + ((Ax_{1,L_{17}} \bmod M) \gg 17) - b_2 2^{31} + c_2 \bmod 91$$

## # Random String Utils (2º Ataque 1º Paso)

- Usando el primer output  $y_1$  con la ecuación

$$y_1 = (x_1 \gg 17) \mod 91 \implies y_1 - 91k_1 = x_{1,U_{31}}$$

- Enumeramos todos los  $\left\lceil \frac{2^{31}}{91} \right\rceil$  posibles valores de  $k_1$  para generar candidatos para  $x_{1,U_{31}}$

## # Random String Utils (2º Ataque 2º Paso)

- Usando los candidatos de  $x_{1,U_{31}}$  obtenidos del primer paso, calcular la parte de la izquierda

$$(y_2 - (((Ax_{1,U_{31}} + C) \bmod M) \gg 17)) \bmod 91$$

- Asociamos el valor candidato de  $x_{1,U_{31}}$  con su valor que estará en  $[0-90]$
- Nota: Cada valor entre  $[0-90]$  tendrá aproximadamente  $2^{31}/91^2$  candidatos de  $x_{1,U_{31}}$



## # Random String Utils (2º Ataque 3º Paso)

- Enumerar todos los posibles  $2^{17}$  candidatos para  $x_{1,L_{17}}$  y computar el valor de la parte de la derecha

$$(((Ax_{1,L_{17}} \bmod M) \gg 17) - b_2 2^{31} + c_2) \bmod 91$$

- Recordemos que:

$$b_2 \in \{0, 1\} \quad c_2 \in \{0, 1\}$$

- Al igual que en la “parte izquierda”, el resultado serán valores entre 0 y 90
- En total hay  $2^{17} \times 2 \times 2 = 2^{19}$  valores calculados

## # Random String Utils (2º Ataque 4º Paso)

- Para cada valor de la parte derecha que hemos calculado en el tercer paso, buscamos el valor adecuado de los valores calculados de la izquierda
- Aproximadamente tenemos una lista de  $2^{31}/91^2$  candidatos para  $x_{1,U_{31}}$  para buscar su “match” correcto en la lista de  $2^{17}$  candidatos de  $x_{1,L_{17}}$
- Este enfoque de ataque se llama *meet-in-the-middle*
- Básicamente es una compensación de tiempo de computación por memoria

$$2^{19} \times 2^{31}/91^2 \approx 2^{37}$$

## # Random String Utils (2º Ataque EXTRA)

- Para el ataque anterior hemos utilizado  $y_1$  y  $y_2$ 
  - Recordemos que  $y_i$  es el output  $i$  de `nextInt(91)`
- Si incluyéramos también  $y_3$  podríamos reducir la complejidad a:

$$2^{21} \times 2^{31} / 91^3 \approx 2^{32.5}$$

# # Ejemplo



# # SecureRandom( )

- Funciona acumulando entropía de una fuente que son difícilmente observables
- `/dev/urandom` y `/dev/random` en Linux recogen información como interrupciones del hardware como discos duros buscando datos, presiones en el teclado, paquetes en red...
- El objetivo de `SecureRandom()` es producir números de forma no determinista usando distintas fuentes de datos
- Como contrapartida, es mucho más lenta que `Random()`

```
> hexdump -C -n 256 /dev/urandom
00000000 94 eb c8 0c 60 5b ed b7 f6 de e5 d9 4c 5c c8 6c .....`[.....L\..l
00000010 f2 58 0c 45 45 2d 77 7d b4 d7 d2 42 eb b8 99 2b .X.EE-w} ... B ... +
00000020 93 2a a9 c4 3f 0a d6 82 a7 f4 3b 74 ff d6 e8 67 .*..?.....;t...g
00000030 a2 d8 2f 40 59 79 64 f9 27 04 66 72 1c 86 03 01 .. /@Yyd.'fr....
00000040 9d 43 97 12 62 8f 5f 35 3e c8 81 1c 90 b6 17 95 .C..b._5>.....
00000050 23 75 ad e4 44 5a 29 44 b7 b5 8c db 38 09 36 6a #u..DZ)D....8.6j
00000060 6c 1a e6 cf 67 03 c8 e3 f1 86 a7 d8 17 8a 1d a5 l ... g.....
00000070 01 bc 4d 76 9f 4e 25 5c 3b 8b 52 28 d3 07 c1 ad .. Mv.N%\;.R(....
00000080 89 91 69 3f 00 2e f3 16 f5 c0 96 a2 c3 b6 7b ef ..i?.....{.
00000090 6c 99 7c f5 d0 9e b9 07 d3 11 b9 95 b2 2f cd d2 l.|...../..
000000a0 0e 88 a8 3f 30 0d 75 34 a8 ce bb 7e d9 f8 f2 72 ... ?0.u4 ... ~ ... r
000000b0 20 2e fa 0a f5 f8 f5 1c c0 81 f5 0b a4 d8 1c b2 .....
000000c0 8f 34 be 31 7b a0 79 96 de 0b 02 f9 42 61 b1 5f .4.1{.y.....Ba._
000000d0 05 4c 7d 5d bf 2c e8 2e b4 bf 1b cf 90 fa eb ea .L}].,.....
000000e0 40 2f 65 f8 5b 8f cd 23 6c ee 6d ab fc ad 18 b0 @/e.[ .. #l.m....
000000f0 c5 99 dd 9f 4c 7f e2 da a5 38 7e 4e 48 48 28 59 ....L....8~NHH(Y
```

# # Resultados

$2^{48}$	$2^{40.5}$	$2^{32.5}$
130 horas (5 días)	1.5 horas	<1 minuto

Tiempo para conseguir la semilla en cada ataque explicado

# # Conclusiones

- El uso de `java.util.Random` y de `RandomStringUtils` no se debe usar en contextos de seguridad
- La implementación es un Sistema Congruente Lineal, que resulta en generación de números *pseudoaleatorios* de forma determinista
- Hay varios ataques que han conseguido decrementar drásticamente el tiempo necesario para extraer la semilla y predecir futuros outputs
- Happy Hunting!

# Breaking Random( ) Internals For Fun!

Ismael Gómez Esquilichi