Università degli studi di Catania

DIPARTIMENTO DI MATEMATICA E INFORMATICA

Corso di Laurea in Informatica Magistrale

Progetto Computer Security 9CFU

Java **R**andom **N**umber **G**enerator

Studente:
Cristina Parasiliti Parracello
Matricola W82000029

Docente:
Prof. Giampaolo Bella

Anno 2013/2014

Importanza dei numeri Random

Con il termine **RANDOM** si intende un numero casuale imprevedibile, dove ogni valore possibile è equiprobabile, senza nessuna dipendenza tra i numeri generati successivamente.

I numeri random vengono utilizzati in molte applicazioni, tra le più "banali" come i videogiochi, a quelle più complesse come nell'ambito della **crittografia**(chiavi di sessione, nonce).

Per ottenere numeri realmente casuali bisogna introdurre input non deterministici ed ottenere ciò è abbastanza dispendioso a livello temporale, basti pensare ai dispositivi hardware che rilevano fenomeni dal mondo reale (come il rumore atmosferico) per poi utilizzare questi dati come entropia per generatori di numeri casuali.

Per questo si fa sempre più uso di numeri pseudo-casuali.

Essi sono stati ideati in modo che la loro produzione è statisticamente indistinguibile da veri numeri casuali.

Pseudo-random Number Generator

I generatori di numeri pseudo-random (**PRNG**) prendono come input un valore iniziale detto "seed" e mediante un processo deterministico restituiscono un valore pseudo-casuale.

Da qui si deduce che l'introduzione dello stesso seed restituisce lo stesso output.

Pseudo-random Number Generator in java

In Java il PRNG è definito dalla classe ***Java.util.Random***.

I costruttori della classe sono definiti come segue:

```
public  Random() {
            this(++seedUniquifier + System.nanoTime());
}
private static volatile long seedUniquifier = 8682522807148012L;
public  Random(long seed) {
    this.seed = new AtomicLong(0L);
    setSeed(seed);
 }
```

La classe Random prende come input un seed/valore iniziale a 48 bit e come è possibile vedere, se non viene passato nessun seed iniziale dall'utente stesso , la classe ne crea uno da se utilizzando come entropia la System.nanoTime().

Entrambi i costruttori richiamano al suo interno il metodo *setSeed()*, non fa altro che riseminare il seed passato come input nel seguente modo:

```
synchronized public void  setSeed(long seed) {
    seed = (seed ^ multiplier) & mask;
     this.seed.set(seed);
}
```

(i valori delle variabili multiplier e mask sono specificati sotto).

La classe Random implementa al suo interno un generatore congruenziale lineare (**LCG**) , un algoritmo per la generazione di numeri casuali molto conosciuto.

La formula su cui si basa questo algoritmo è la seguente:

$$\mathbf{X_{n+1}=(X_n*a+c) \bmod m}$$

dove $\mathbf{X_n}$ è un valore della successione dei numeri pseudo casuali generati

- **a** è il moltiplicatore
- **c** è l'incremento
- $\mathbf{X_0}$ è il seed/valore iniziale
- **m** è il modulo e rappresenta anche il **periodo** del generatore, cioè date le m chiamate al generatore, alla m+1 chiamata ritornerà uno tra i valori già restituiti precedentemente, con la stessa probabilità.

Nella classe Random questi valori sono definiti dalle seguenti variabili:
- *private final static long **multiplier** = 0x5DEECE66DL;*
- *private final static long **addend** = 0xBL;*
- *private final static long **mask** = (1L << 48) - 1;*

(la mask corrisponde al % 48 poiché non fa altro che troncare a 48 bit qualsiasi valore).

La classe fornisce i seguenti metodi:

-nextInt()

-nextInt(long int)

-nextBoolean()

-nextBytes(byte [] bytes)

-nextDouble()

-nextFloat()

-nextGaussian()

-nextLong()

Ognuno di essi restituisce il prossimo valore pseudo-random.

Essi richiamano al suo interno il metodo *next(int bits)* dove è definito il generatore LCG, come segue:

```
protected int  next(int bits) {
        long oldseed = 0, nextseed = 0;
        AtomicLong seed = this.seed;
```

```
while (!seed.compareAndSet(oldseed, nextseed)){

        oldseed = seed.get();
        nextseed = (oldseed * multiplier + addend) & mask;

    }
     return (int)(nextseed >>> (48 - bits));
}
```

È facile intuire che questo metodo è basato su un algoritmo deterministico, in quanto basta conoscere il valore della variabile oldseed per risalire al nextseed, cioè al prossimo valore pseudo-random che verrà generato.

Dimostrazione della non sicurezza della classe *java.util.Random*

È possibile dimostrare che la classe Random non è sicura, analizzando per esempio il metodo nextInt().

Come è possibile vedere dalla sua implementazione

```
public int  nextInt() {
        return next(32);
    }
```

non fa altro che richiamare al suo interno il metodo *next()* e restituire un valore pseudorandom a 32 bit.

Come descritto sopra , il metodo *next()* calcola il nextseed tramite la formula LCG e ritorna il valore nextseed shiftato di (48-32) bit.

1°CASO: l'utente passa il seed iniziale al momento della creazione dell'oggetto Random.

In questo caso si hanno a disposizione tutti i valori delle variabili, basta quindi calcolare il seed iniziale settato dal metodo *setSeed*() per poi utilizzare esso come oldseed alla formula specificata in *next(),* e rifare lo stesso ragionamento per i prossimi valori pseudorandom che si vogliono calcolare.

```
public static void main(String[] args) {
  //creo un'istanza Random passando come input il valore 2
```

```
    Random random=new Random(2);


    //calcolo ciò che fa il metodo setSeed()
    long seedIniziale=(2^multiplier)&mask;

    // calcolo il 1° nextseed che mi ritorna il metodo nextInt()
    int x1=(int) (((seedIniziale*multiplier + addend)&mask)>>>16);

  //utilizzo  x1(a 48 bit) come input per calcolare il 2° nextseed
   int x2=(int) ((((((seedIniziale* multiplier + addend)&mask)*multiplier +
  addend))&mask)>>>16);

//utilizzo x2(a 48 bit)=tmp come input per calcolare il 3° nextseed
long tmp=(((((seedIniziale* multiplier + addend)&mask)*multiplier +
addend))&mask);
int x3=(int)(((tmp*multiplier+addend)&mask)>>>16);


//richiamo adesso il metodo 3 volte
    int v1=random.nextInt();
    int v2=random.nextInt();
    int v3=random.nextInt();


System.out.println("il primo valore random calcolato da me è:"+x1);
System.out.println("il primo valore random ritornato dal  metodo è:"+v1);
System.out.println("il secondo valore random calcolato da me è: "+ x2);
System.out.println("il secondo valore random ritornato dal metodo è:"+v2);
System.out.println("il terzo valore random calcolato da me è: "+  x3);
System.out.println("il terzo valore random ritornato dal metodo è: "+v3);

}
```

Quello che viene restituito in output è il seguente:

```
il primo valore random calcolato da me è: -1154715079
il primo valore random ritornato dal metodo è: -1154715079
il secondo valore random calcolato da me è: 1260042744
il secondo valore random ritornato dal metodo è: 1260042744
il terzo valore random calcolato da me è: -423279216
il terzo valore random ritornato dal metodo è: -423279216
```

Come è possibile notare i valori calcolati e quelli che restituiti dal metodo *nextInt()* coincidono.

2°CASO: l'utente non passa il seed iniziale al momento della creazione dell'oggetto Random.

In questo caso non si hanno a disposizione tutti i valori delle variabili, poiché il seed iniziale è creato utilizzando entropia imprevedibile.
Si possono però calcolare quali sono i successivi valori pseudorandom che il metodo nextInt() restituisce, richiamando il metodo su un'istanza Random due volte ed analizzando i due valori restituiti.

Alla prima chiamata il metodo restituisce il valore x1 a 32 bit.
Alla seconda chiamata il metodo restituisce il valore x2 a 32 bit.
I due valori v1 e v2 sono "legati" dalla formula ,poiché come specificato nel metodo next() , v2 è calcolato nel seguente modo:

→ v2=(((v1')*multiplier+addend)& mask)>>>16
dove v1' non è altro che v1 a 48 bit.

Le uniche informazioni che si hanno sono:

- v1=((seediniziale*multiplier+addend)& mask)>>>16 (32 bit)
- v1'=v1*2^{16} possibili valori (in quanto v1' è v1 a 48 bit).

Basta allora provare tutti i possibili valori di v1' e verificare se vi è un matching, nel seguente modo:

NOTA: v1' corrisponde al seed.

```
public static void main(String[] args) {
    Random random=new Random();
    long v1=random.nextInt();
    long v2=random.nextInt();

    for (int i=0; i<65536;i++){
    long seed=v1*65536+i;
    if ((((seed * multiplier + addend) & mask) >>> 16) == v2) {

    System.out.println("trovato matching");

    //calcolo le random successive

    //v3=((v2'*multiplier+addend)&mask))>>>16 ,v2' a 48 bit
    long tmp=((seed * multiplier + addend) & mask); //tmp=v2'
    long x3=((tmp * multiplier + addend)& mask) >>>16;
```

```
    //v4=((v3'*multiplier+addend)&mask))>>>16, v3' a 48 bit
    long tmp2=((tmp * multiplier + addend)& mask); //tmp2=v3'
    int x4=(int) (((tmp2 * multiplier + addend)& mask)>>>16);


     //v5=((v4'*multiplier+addend)&mask))>>>16, v4'a 48 bit
    long tmp3=((tmp2 * multiplier + addend)& mask); tmp3=v4'
    int x5=(int) (((tmp3* multiplier + addend)& mask)>>>16);

System.out.println("il prossimo valore random calcolato da me è "+x3);
System.out.println("il prossimo valore random calcolato da me è "+x4);
System.out.println("il prossimo valore random calcolato da me è "+x5);



int v3=random.nextInt();
int v4=random.nextInt();
int v5=random.nextInt();

System.out.println(v3);
System.out.println(v4);
System.out.println(v5);

}
}
}
```

In output i valori calcolati (x_i) corrispondo a quelli restituiti dal metodo (v_i).

Entrambi i casi verificano che la classe Random non è sicura , in quanto è facile determinare i numeri successivi pseudo-casuali.
Questo è dovuto al fatto che la classe Random utilizza un algoritmo deterministico per calcolare valori pseudo-casuali.

Un ragionamento simile è possibile farlo anche per tutti gli altri metodi che la classe fornisce , poiché come specificato sopra, tutti i metodi fanno riferimento al metodo *next()*.

Cryptographically Secure Pseudo-random Number Generator

In java un **CSPRNG** è implementato dalla classe *java.security.SecureRandom*.

Un oggetto SecureRandom è definito da un **algoritmo**, un **provider** e una **SPI** (Service Provider Interface).
In una Service Provider Interface sono definiti i metodi astratti:
- *engineGeneratorSeed()*
- *engineSetSeed()*
-*engineNextBytes()*
i quali devono essere implementati da ogni provider che vuole fornire un PRNG crittograficamente forte.

Lo schema generale che SecureRandom segue è il seguente:

seed →setSeed() →engineSetSeed() →PRNG—>next() →data

- Il **seed** o viene dato come input quando viene creata un'istanza SecureRandom o viene generato dalla classe stessa.

- ***setSeed(seed)*** richiama al suo interno *engineSetSeed()* che prende l'input seed ed esso viene integrato durante la risemina del seed.

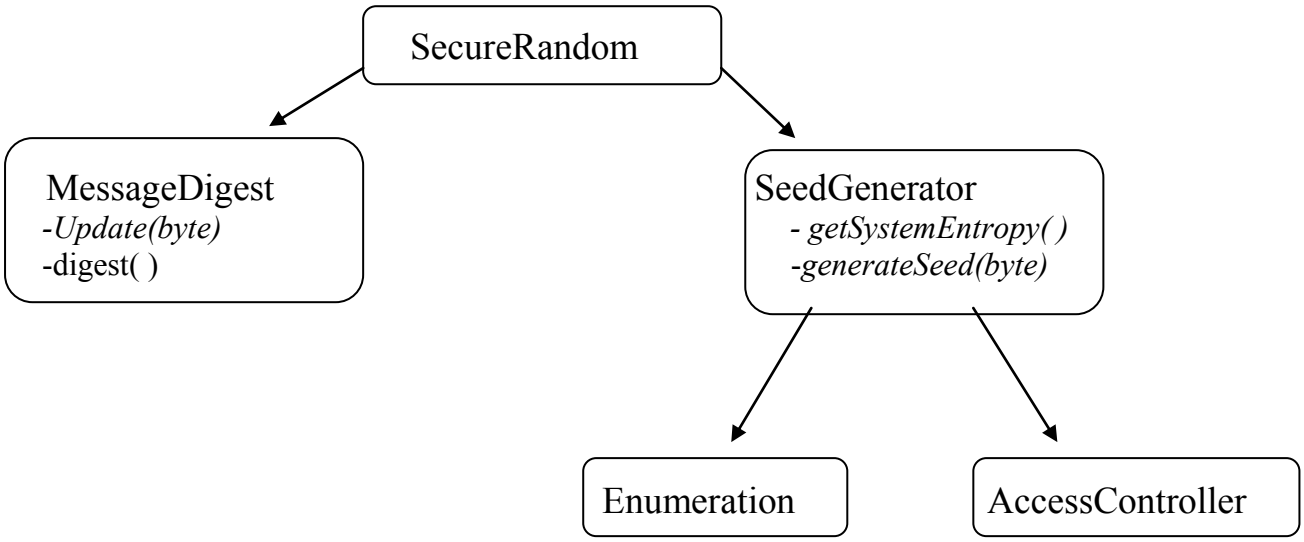
- Questo seed di output viene passato all'algoritmo **PRNG** scelto o dato di default (SHA1PRNG nei sistemi Windows, NativePRNG nei sistemi Linux/Mac) e integrato con l'entropia raccolta dall'algoritmo.

- Infine preso questo valore restituito dall'algoritmo ,si passa come seed iniziale ad uno dei metodi della classe **java.util.Random** che si sceglie, per ottenere il valore pseudo-random.

Analisi della classe java.security.provider.SecureRandom

Java di default assegna come algoritmo **SHA1** nei sistemi Windows, come provider la **SUN** e come SPI quella della Sun.

Questa classe dal punto di vista strutturale è abbastanza complessa, poiché utilizza metodi e oggetti di altre classi, come è possibile vedere dallo schema:



- La classe **MessageDigest** fornisce applicazioni di funzionalità ad algoritmi message digest come MD5 o SHA.
  -il metodo *update()* non fa altro che aggiornare il digest.
  -il metodo *digest()* completa il calcolo hash eseguendo operazioni come il padding.

- La classe **SeedGenerator** genera seed per CSPRNG. Il seed viene prodotto o tramite il calcolo di attività corrente del sistema o da un dispositivo di raccolta entropia.
  -il metodo *getSystemEntropy()* recupera alcune informazioni di sistema (il tempo di sistema in millisecondi, IP address, la directory temporanea, stato della memoria).
  -il metodo *generateSeed(byte)* genera un seed tramite il valore dato in inp*ut.*

La classe SecureRandom è definita un **CSPRNG** poiché al suo interno raccoglie ed utilizza entropia imprevedibile, per generare il seed da fornire come input iniziale ad uno dei metodi della classe Random.
Tramite il metodo *engineNextByte()* (definito dal provider SUN) recupera entropia e crea il seed come segue:

```
public synchronized void engineNextBytes(byte[] result) {
...

//lo state==null se non è stato passato nessun seed dall'utente al    momento della crezione dell'oggetto SecureRandom

if (state == null) {

//il seeder==null se non è mai stato chiamato prima il metodo

  if (seeder == null){
seeder= new SecureRandom(SeedGenerator.getSystemEntropy());

//genera un array di  byte della dimensione del digest contenente i dati casuali forniti dall'entropia
seeder.engineSetSeed(engineGenerateSeed(DIGEST_SIZE));

 }

//altrimenti se il metodo è già stato chiamato almeno una volta, richiama il metodo sul seed restituito dalla chiamata precedente.

        byte[] seed = new byte[DIGEST_SIZE];
        seeder.engineNextBytes(seed);
        state = digest.digest(seed);
    }
...
```

Il metodo *getSystemEntropy()* raccoglie dati del sistema e aggiorna man mano l'oggetto MessageDigest tramite il metodo *update(),*per poi restituire tutta l'entropia raccolta.

```
static byte[] getSystemEntropy()
      byte[] ba;
    final MessageDigest md;

    try {
       md = MessageDigest.getInstance("SHA");
```

```
        }
           catch (NoSuchAlgorithmException nsae) {
          throw new InternalError("internal error: SHA-1 not       available.");
        }

        // aggiorna md con il tempo corrente del sistema in millisecondi
        byte b =(byte)System.currentTimeMillis();
        md.update(b);

        java.security.AccessController.doPrivileged
            (new java.security.PrivilegedAction<Void>() {
               public Void run() {

       try {
          String s;

        //restituisce le proprietà del sistema
       Properties p = System.getProperties();

      //Restituisce una enumerazione di tutte le chiavi in questo elenco
        proprietà
      Enumeration<?> e = p.propertyNames();


    //fino a quando questa enumerazione contiene elementi
      aggiorna md  con i suoi successivi elementi
       while (e.hasMoreElements()) {
          s =(String)e.nextElement();
                md.update(s.getBytes());

    //p.getProperty(s) ritorna il valore di s in questo elenco
                md.update(p.getProperty(s).getBytes());

        }

  // aggiorna md con l'IP address
  md.update(InetAddress.getLocalHost().toString().getBytes());

   // aggiorna md con la directory temporanea
      File f = new File(p.getProperty("java.io.tmpdir"));
      String[] sa = f.list();
                    for(int i = 0; i < sa.length; i++)
                       md.update(sa[i].getBytes());
```

```
            } catch (Exception ex) {
                md.update((byte)ex.hashCode());
            }

// aggiorna md con lo stato della memoria
  Runtime rt = Runtime.getRuntime();
 byte[] memBytes = longToByteArray(rt.totalMemory());
            md.update(memBytes, 0, memBytes.length);
            memBytes = longToByteArray(rt.freeMemory());
            md.update(memBytes, 0, memBytes.length);

            return null;
          }
        });
    return md.digest();
  }
```

Nei sistemi Linux/Mac java assegna come algoritmo di default **NativePRNG**.

L'algoritmo è basato sulla lettura di file speciali quali */dev/random* e */dev/urandom* che restituiscono numeri casuali, basati su eventi imprevedibili quali gli orari in cui si verificano interrupt sui disk-drive, eventi di mouse e tastiera . L'intento è quello di raccogliere i dati in modo che un aggressore remoto o locale non può prevedere.