Curso: (30227) Seguridad Informática

Fernando Tricas García

Departamento de Informática e Ingeniería de Sistemas
Universidad de Zaragoza
http://webdiis.unizar.es/~ftricas/
http://moodle.unizar.es/
ftricas@unizar.es

Tema Aleatoriedad y determinismo

Fernando Tricas García

Departamento de Informática e Ingeniería de Sistemas
Universidad de Zaragoza
http://webdiis.unizar.es/~ftricas/
http://moodle.unizar.es/
ftricas@unizar.es

Aleatoriedad y determinismo

La aleatoriedad se utiliza para que determinadas componentes del sistema no sean predecibles:

- Identificadores, fundamentalmente
  - En URLs
  - En gestión de sesiones
  - ...
- Y claves

Hace falta:

- Longitud adecuada
- Con suficiente impredecibilidad

Departamento de Informática e Ingeniería de Sistemas
Universidad Zaragoza

Aleatoriedad y determinismo

- Mal uso de los sistemas de generación de números aleatorios: problema de seguridad
- `random()` y similares no ofrecen números verdaderamente aleatorios (PRNG)
- Los ataques son difíciles, pero no tanto como solemos creer
- Los computadores son especialmente 'malos' para estas cosas
- Vamos a discutir el uso de números seudo-aleatorios de forma adecuada

Generadores de números pseudo...

- Los computadores son completamente deterministas
- Se utilizan algoritmos generadores (*pseudo random number generators – PRNG*)
  A partir de una semilla (un número inicial) se calculan los siguientes.
  - Si las semillas se pueden adivinar, los números también
  - Hay que tratar de tener semillas imposibles de calcular o de predecir
  - A veces es útil que esto sea así (depuración, simulación, ...)

Generadores ...

- Siempre que sea posible, semillas generadas 'más aleatoriamente'
- Si la semilla tiene n bits 'buenos', el ataque necesitará 2^n operaciones
- Hace falta conocer el algoritmo, no siempre es así, pero la experiencia demuestra una y otra vez, que es mejor suponer que se conoce (recordar: defensa en profundidad)
- Se trata de suposiciones parecidas a las que hacen los criptógrafos. (se consideran así)

Dos tipos de algoritmos

- Estadísticos.
  1. Están pensados pasar los tests de aleatoriedad estadística, lo que no significa que sean impredecibles (sólo que se distribuyen razonablemente).
  2. Un objetivo importante es la reproducibilidad, con una semilla al principio

Algoritmos criptográficos

- Criptográficos
  - Necesitamos más: que sean seguros criptográficamente (los estadísticos son de uso frecuente)
  - Su seguridad depende de la entropía de la semilla
  - `random()`, `rand()`, `drand48()`, `lrand48()`, `mrand48()` no son criptográficos
    - `java.util.Random` tampoco.

Ejemplos

El más frecuente, generador lineal basado en congruencias

$$X_{n+1} = (aX_n + b) \text{mod } c$$

Valores adecuados de a, b, y c proporcionan buenos resultados para aplicaciones estadísticas

Departamento de Informática e Ingeniería de Sistemas
Universidad Zaragoza

Más detallado

```
long long          RandSeed = #### ;
unsigned long
Random(long max){
   long long x;
   double              i;
   unsigned long       final;

   x = 0xffffffff;
   x += 1;

   RandSeed *= ((long long)134775813);
   RandSeed += 1;
   RandSeed = RandSeed %x;

   i = ((double)RandSeed) / (double)0xffffffff;

   final = (long)(max * i);

   return (unsigned long)final;
}
```

Generadores basados en congruencias

- Generan un número entre 0 y 1, o un entero equiprobable
- Son muy fáciles de atacar, porque la mayoría usan valores de 32 bits
- Observando los números generados, se puede adivinar la semilla (sólo hay 4.294.867.295 posibilidades)
- Aumentando el número de bits no mejora, porque también hay otros ataques posibles
- ¡Usar uno criptográfico!

Blum-Blum-Shub

- Método criptográfico, basado en la dificultad de factorizar primos grandes
- No muy práctico
- Importante porque se basa en principios matemáticos simples
  1. Dos números primos grandes p y q
     (p mod 4 = 3, q mod 4 = 3)
     Secretos.
  2. N = p x q es el número de Blum
  3. Elegir una semilla aleatoria s (entre 1 y N, que no sea p ni q)
  4. $x_0 = s^2 \bmod N$
  5. $x_i = x_{i-1}^2 \bmod N$
  6. $b_i = x_i \bmod 2$
     El bit menos significativo de x_i se usa como bit aleatorio.

Departamento de Informática e Ingeniería de Sistemas
Universidad Zaragoza

Ataques

- Criptoanálisis
- Conocimiento que se pueda tener sobre el estado interno del PRNG

¿Dónde conseguir entropía?

- La semilla es muy importante
  - No se puede incluir en el código, ni pedir que alguien la teclee
  - No usar direcciones de red, nombres de máquinas, de gente, de la madre ...
  - El reloj tampoco es una buena fuente: en la mayoría de los casos, 32 bits, pero muchas veces, ni siquiera eso.

¿Dónde conseguir entropía?: hw

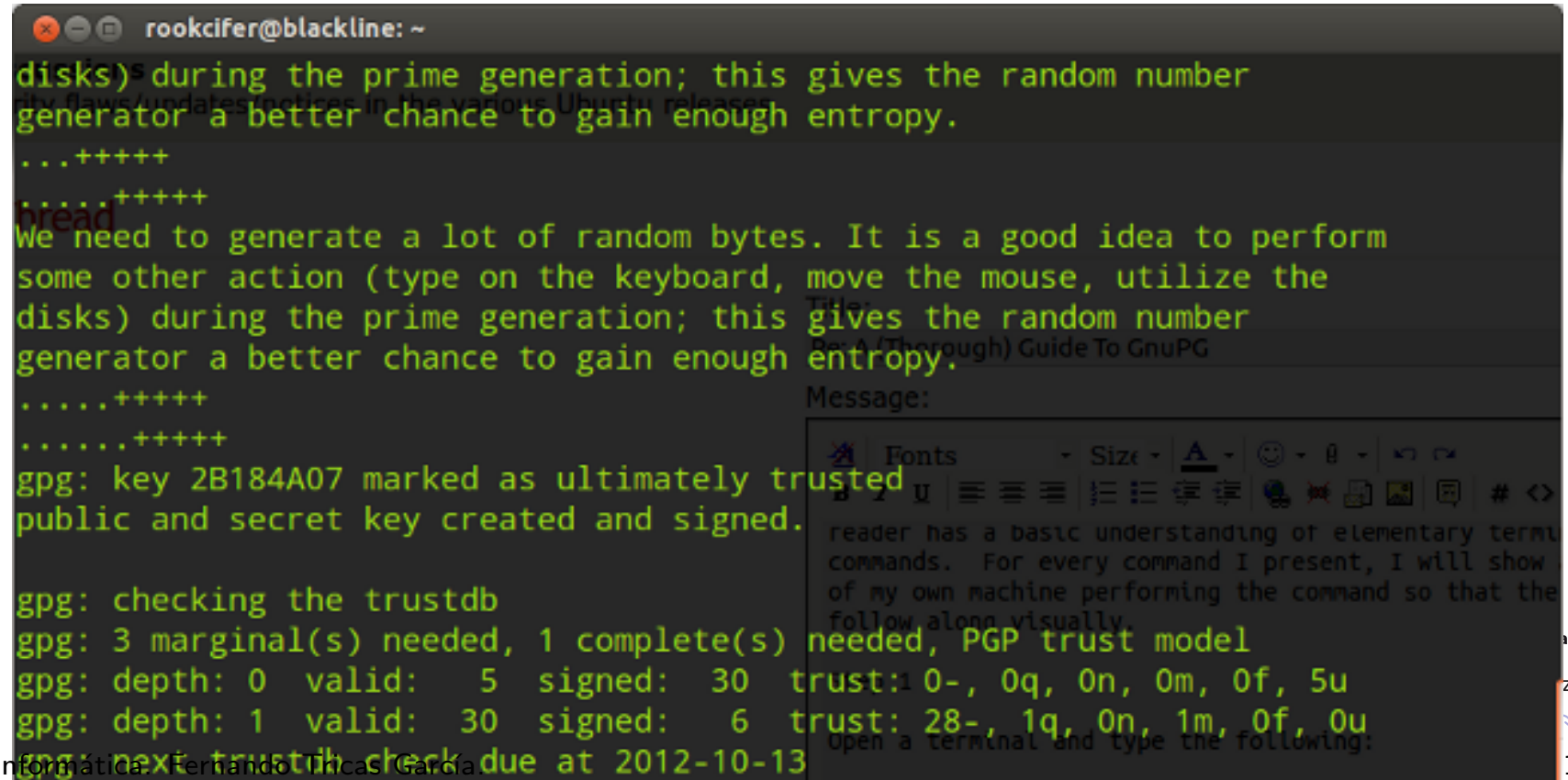
- La mejor solución es el 'hardware' específico
- No siempre es factible
- Ejemplos:
  - Medir el ruido térmico en un diodo semiconductor
  - Contador Geiger que emite un pulso cada vez que se detecta una bajada de radioactividad, el intervalo temporal es aleatorio
- Conviene procesar después (y vigilar).

Departamento de Informática e Ingeniería de Sistemas
Universidad Zaragoza

¿Dónde conseguir entropía?: sw

- Casi siempre se usan de este tipo
- Se supone que la máquina no ha sido comprometida
- Se buscan fuentes de aleatoriedad:
  - Muestreo del teclado o del ratón
  - Por ejemplo: 'mueva el ratón, o teclee un texto'

Visto en `http://ubuntuforums.org/showthread.php?t=1975929`



¿Dónde conseguir entropía?

- Cuidado con el ratón: la información viaja por la red y es visible para las aplicaciones
- Cuidado con el teclado: auto-repetición
- Tiempo para la finalización de alguna tarea
  1. Tiempo tardarmos en conseguir tiempo de procesador un determinado número de veces.
  2. Tiempo que se tarda en lanzar un hilo que no hace nada, ...
- Variaciones entre el reloj y la generación de interrupciones
- Tráfico de la red, tiempo de búsqueda en el disco (cuidado con este)

¿Dónde conseguir entropía?

- Es difícil saber cómo de buenos son los datos que se consiguen así
  - Se puede tratar de medir, ejecutando repetidamente, y observando qué bits cambian con una probabilidad similar
- Muchas de estas técnicas son susceptibles de recibir ataques, si alguien tiene acceso a la máquina

Algunos (malos) ejemplos

Navegador Netscape

En 1996 demostraron un fallo, al elegir una mala semilla (fácilmente predecible) para usar con el SSL.
Utilizaban la hora del sistema (en sistemas de tipo Unix con una precisión de milisegundos pero en otros sólo de 1/60 segundo o menos:

$$60 \times 60 \times 60 = 216000 \text{ valores posibles})$$

Ian Goldberg and David Wagner,
'Randomness and the Netscape Browser'

`http://www.cs.berkeley.edu/~daw/papers/ddj-netscape.html`

Algunos (malos) ejemplos

Debian, openSSH

En mayo de 2008, un problema en Debian por un generador aleatorio predecible: Sólo $2^{15}(32767)$ claves posibles.
El desarrollador comentó parte del código (que no comprendía bien) porque pensó que era innecesario al ser señalado como tal ('using uninitialized data') por una herramienta de ayuda ('Valgrind').

`http://www.debian.org/security/2008/dsa-1571`
`http://etbe.coker.com.au/2008/05/18/debian-ssh-problems/`

`https://blog.isotoma.com/2008/05/debians-openssl-disaster/`

Departamento de Informática e Ingeniería de Sistemas
Universidad Zaragoza

Algunos (malos) ejemplos

Ruby, openSSH (Nov. 2011)

ext/openssl/ossl_pkey_rsa.c (rsa_generate): [SECURITY] Set RSA exponent value correctly. Awful bug. This bug caused **exponent of generated key to be always '1'**. By default, and regardless of e given as a parameter. !!! Keys generated by this code (trunk after 2011-09-01) must be re-generated !!! (ruby_1_9_3 is safe)

Mal:

```
for (i = 0; i < (int)sizeof(exp); ++i) {
```

Bien (* **8**) :

```
for (i = 0; i < (int)sizeof(exp) * 8; ++i) {
  if (exp & (1 << i)) {
    if (BN_set_bit(e, i) == 0) {
      BN_free(e);
```

"Horrible fallo" de seguridad criptográfico en Ruby

http://unaaldia.hispasec.com/2011/11/horrible-fallo-de-seguridad.html

http://www.h-online.com/security/news/item/Ruby-s-RSA-crypto-bug-near-miss-1374968.html

http://svn.ruby-lang.org/cgi-bin/viewvc.cgi/trunk/ext/openssl/ossl_pkey_rsa.c?r1=33633&r2=33632&pathrev=33633

Errores frecuentes con las semillas

- Semillas con pocos bits (8 bits $\rightarrow$ 256 semillas iniciales)
- Hash de la hora (resolución de 1/60 de segundo $\rightarrow$ el atacante puede conocer la hora $\rightarrow$ 60 x 60 x 60 = 216000 valores)
- Divulgar la semilla: elegir la hora como semilla, y enviarla por correo, por ejemplo

Mas noes

- Direcciones de red, números de serie de hw, ..
- Tiempo de llegada de los paquetes (manipulable)
- Selecciones de datos (por ejemplo una base de datos en CD)
- Un sí `/dev/audio`, comprimido, con cuidado

Algunas sugerencias

- Tiny (EGADS, para Unix y Windows)
- `CryptGenRandom` en Windows es un PRNG criptográfico (pero no ha sido validado por la comunidad criptográfica)
- `/dev/ramdon` es un generador de entropía y `/dev/urandom` es un generador de números seudoaleatorios para Linux y otros sistemas similares
- Cuidado al manejarlos (en 2000 se descubrió un fallo en PGP relacionado con un mal uso de `/dev/ramdon`).
- `http://egd.sourceforge.net/`

  Entropy Gathering Daemon

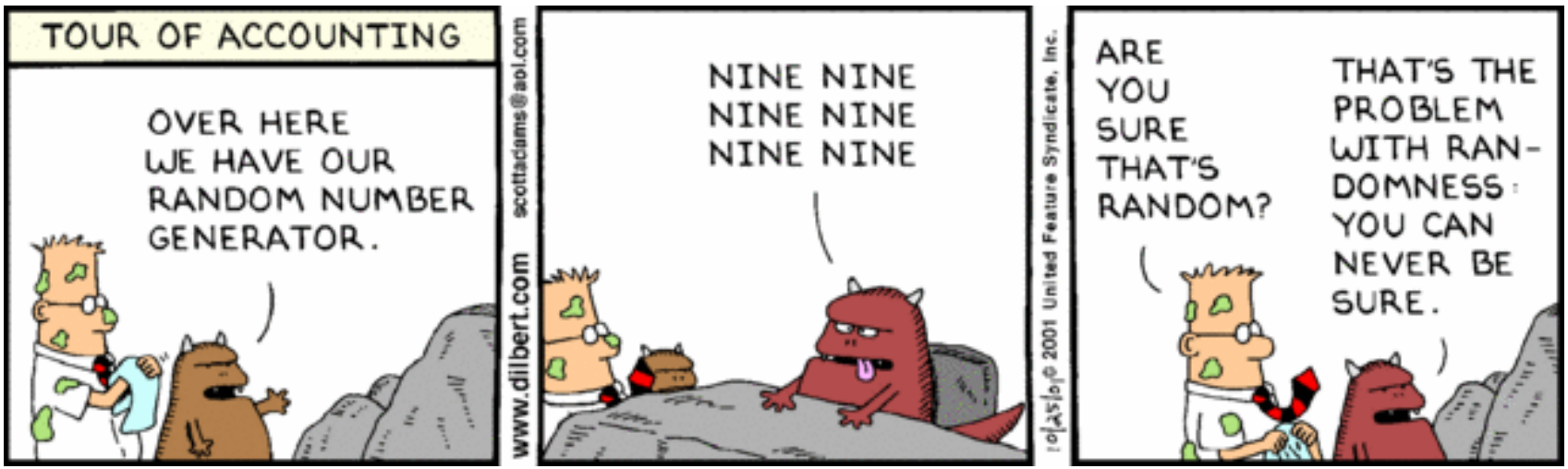
¿Y en Java?

`java.security.SecureRandom`

- Es portable
- No usar `SecureRandom` como semilla para `java.util.Random`, no es lo suficientemente bueno
- `java.util.Rand` es un PRNG clásico
- Si no se le proporciona una semilla, tarda bastante en empezar (hasta 20, aunque puede ser alrededor de 3)

Conclusiones

- Utilizar la mejor fuente de entropía disponible
- Mejor si es hardware
- Tratar de estimar cuanta entropía tenemos



http://dilbert.com/strips/comic/2001-10-25/