random numbers

The generation of random numbers is too important to be left to chance

Robert Coveyou, Oak Ridge National Laboratory

random numbers

- we have seen that in many applications we need random number generators
- for example, we use random number generator to obtain session keys; to avoid key guessing
  - if an adversary sees a sequence of session keys, then he/she must NOT be able to guess next session key (even in a probabilistic way)

- note: good random number generator for cryptography is expensive

generators of randomness

two principal methods

- measure some physical phenomenon that is expected to be random and then compensates for possible biases in the measurement process (random number generator, RNG)
- use algorithms producing long sequences of apparently random results, which are in fact completely determined by an initial value, known as a seed or key (pseudorandom number generator, PRNG)

- deterministic computations cannot give true random numbers because they are inherently predictable
- distinguishing a "true" random number from the output of a pseudo-random number generator is a very difficult problem
  - carefully chosen pseudo-random number generators can be used instead of true random numbers in many applications
  - rigorous statistical analysis of the output is often needed to have confidence in the algorithm

random number generator

- use of systems data
- use of external information
- above data are used to initialize a pseudorandom number generator, a program that:
  - given an initial seed (random)
  - obtains a long sequence of numbers

- if an adversary sees a long sequence of numbers in output it should be computationally very expensive to guess next output number (even in a probabilistic sense)

initial seed

different sources

- machine /network
  - clock
  - free space on disk
  - number of files on disk
  - info on operating system (I/O queues, buffer state etc.)
  - user information (e.g., windows and size of windows)
  - inter-arrival time of packets
- user
  - keyboard & mouse timing

initial seed: how many bits of randomness?

- in many cases, sources provide few bits of randomness (in fact much information can be easily guessed):
  - clock: we can use only low order digit (e.g., milliseconds: 10 bits of randomness)
  - day, hour and minute can be easily guessed and are not random
- it is advantageous to mix several sources of randomness using crypto functions

common mistakes

- using small initial seed
  - ex.: 16 bits seed give only 65536 different seeds and attacker can try them all
- using current time
  - if clock granularity is 10 msec then if we approximately know the time (just the hour) there are
    60 (minutes) × 60 (seconds) × 100 (hundredths of a second) =
    = 360000 different choices only!
- divulging seed value
  - an implementation used the time of day to choose a per-message encryption key; the time of day in this case may have had sufficient granularity, but the problem was that the application included the time of day in the unencrypted header of the message!

Netscape 1.1 seeding & key generation (1996)

```
global variable seed;

RNG_CreateContext()
 (seconds, microseconds) = time of day;
 /* Time elapsed since 1970 */
 pid = process ID; ppid = parent process ID;
 a = mklcpr(microseconds);
 b = mklcpr(pid + seconds + (ppid << 12));
 seed = MD5(a, b);

mklcpr(x) /* not cryptographically significant */
 return ((0xDEECE66D * x + 0x2BBB62DC) >> 1);

RNG_GenerateRandomBytes()
 x = MD5(seed);
 seed = seed + 1;
 return x;
```

MD5 (Message-Digest 5, by Ron Rivest, '91)

- widely used cryptographic hash function with a 128-bit hash value
  - an MD5 hash is typically expressed as a 32-digit hexadecimal number
- internet standard (RFC 1321)
- employed in a wide variety of security applications, and is also commonly used to check the integrity of files
- **not collision resistant**
  - not suitable for applications like SSL certificates or digital signatures that rely on this property

about collision weakness of MD5

in 1995, collisions were found in the compression function of MD5, and Hans Dobbertin wrote in the RSA Laboratories technical newsletter, "The presented attack does not yet threaten practical applications of MD5, but it comes rather close ... in the future MD5 should no longer be implemented...where a collision-resistant hash function is required."

in 2005, researchers were able to create pairs of PostScript documents and X.509 certificates with the same hash. Later that year, Ron Rivest wrote, "MD5 and SHA1 are both clearly broken (in terms of collision-resistance)," and RSA Laboratories wrote that "next-generation products will need to move to new algorithms."

on 30 December 2008, a group of researchers announced that they had used MD5 collisions to create an intermediate certificate authority certificate which appeared to be legitimate when checked via its MD5 hash

assessing randomness in Netscape 1.1

- if attacker does not have an account on the attacked UNIX machine pid and ppid are unknown
- even though the pid and ppid are 15 bit quantities on most UNIX machines, the sum pid + (ppid << 12) has only 27 bits, not 30
- if the value of seconds is known, a has only 20 unknown bits, and b has only 27 unknown bits. This leaves, at most, **47 bits of randomness** in the secret key
- in addition, ppid & pid can be hacked
  - ppid is often 1 (for example, when the user starts Netscape from an X Window system menu); if not, it is usually just a bit smaller than the pid
  - mail-transport agent sendmail generates Message-IDs for outgoing mail using its process ID; as a result, sending e-mail to an invalid user on the attacked machine will cause the message to bounce back to the sender; the Message-ID contained in the reply message will tell the sender the last process ID used on that machine. If the user started Netscape relatively recently, and that the machine is not heavily loaded, this will closely approximate Netscape's pid

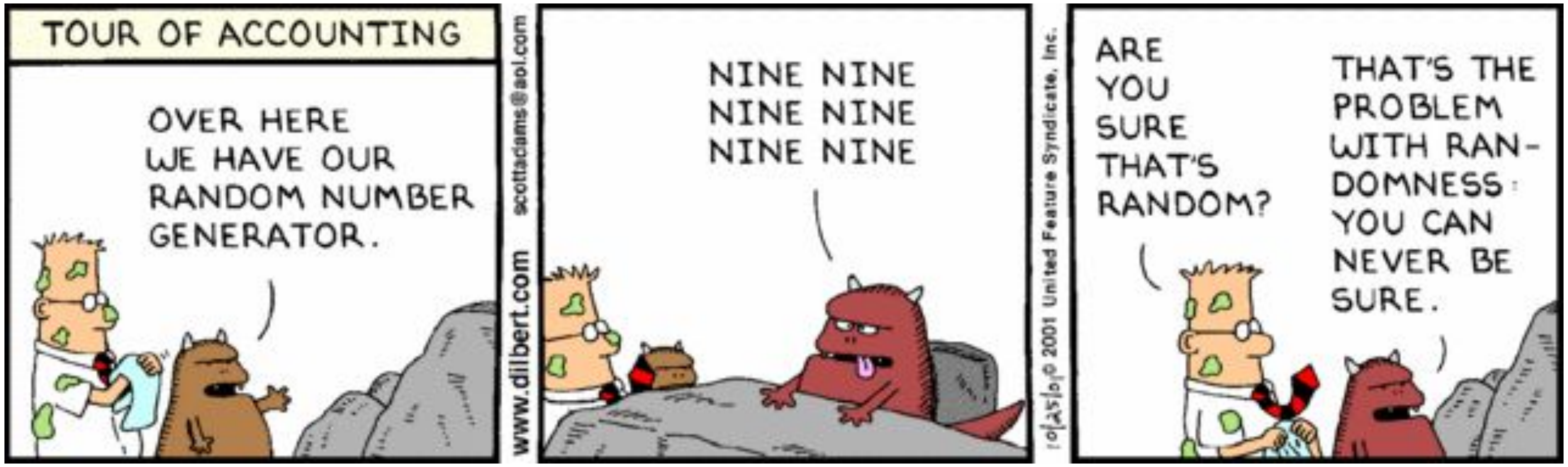
random number bug in Debian Linux (2006)

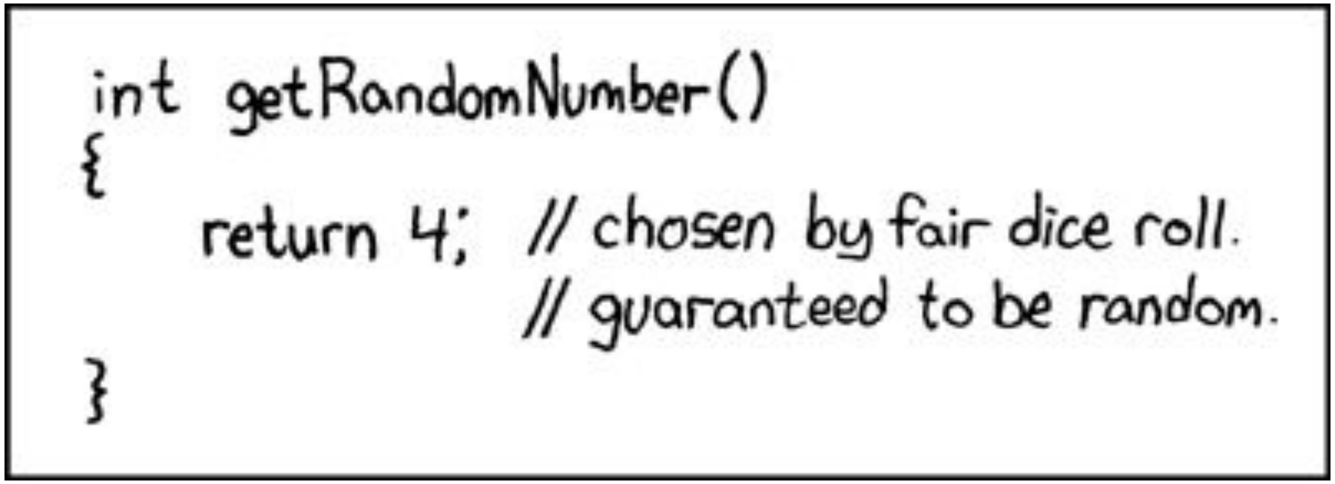
- announced in 2008, affecting Debian + *Ubuntu
  - http://www.debian.org/security/2008/dsa-1571
- vulnerability in the OpenSSL package, caused by the removal of the following two lines of code from **md_rand.c**

```
MD_Update(&m,buf,j);
[ .. ]
MD_Update(&m,buf,j); /* purify complains */
```

- lines removed because they caused Valgrind (OS debugging tool) and Purify (a proprietary debugger, from IBM) to warn about use of uninitialized data in any code linked to OpenSSL
- instead of mixing in random data for the initial seed, the only used "random" value was the current process ID
  - on the Linux platform, the default maximum process ID is 32768, resulting in a very small number of seed values being used for all PRNG operations

random generators' folklore


TOUR OF ACCOUNTING
OVER HERE WE HAVE OUR RANDOM NUMBER GENERATOR.
www.dilbert.com scottadams@aol.com
NINE NINE NINE NINE NINE NINE
10/25/01 © 2001 United Feature Syndicate, Inc.
ARE YOU SURE THAT'S RANDOM?
THAT'S THE PROBLEM WITH RANDOMNESS: YOU CAN NEVER BE SURE.



```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

Bruce Schneier's commentary

from Crypto-Gram Newsletter, June 15, 2008

Random numbers are used everywhere in cryptography, for both short- and long-term security. And, as we've seen here, security flaws in random number generators are really easy to accidentally create and really hard to discover after the fact. Back when the NSA was routinely weakening commercial cryptography, their favorite technique was reducing the entropy of the random number generator.

BSI evaluation criteria

- Bundesamt für Sicherheit in der Informationstechnik (BSI - in English: Federal Office for Information Security)
- defines four criteria for quality of deterministic random number generators, named K1, K2, K3 and K4
- K1 — A sequence of random numbers with a low probability of containing identical consecutive elements (runs).
- K2 (see next slide)
- K3 — It should be impossible for attacker to calculate/guess, from any given sub-sequence, any previous or future values in the sequence, nor any inner state of the generator
- K4 — It should be impossible for attacker to calculate/guess from an inner state of the generator, any previous numbers in the sequence or any previous inner generator states
- for cryptographic applications, only generators meeting the K4 standard are acceptable

K2 class

- the generated sequence of numbers is indistinguishable from "true random" numbers according to specified statistical tests
- monobit test
  - equal numbers of ones and zeros in the sequence
- poker test
  - a special instance of the X^2 (chi-square) test
- runs test
  - counts the frequency of runs of various lengths
- longruns test
  - checks whether there exists any run of length 34 or greater in 20 000 bits of the sequence
- autocorrelation test
- these requirements are a test of how well a bit sequence:
  - has zeros and ones equally often;
  - after a sequence of n zeros (or ones), the next bit a one (or zero) with probability one-half;
  - and any selected subsequence contains no information about the next element(s) in the sequence.

cryptographically secure pseudorandom number generators (CSPRGN)

meet requirements of ordinary PRNG, and

- next-bit test
  - given the first k bits of a random sequence, there is no polynomial-time algorithm that can predict the (k+1)th bit with probability of success better than 50%
- withstand "state compromise extensions"
  - if part or all its state has been revealed (or guessed correctly), it should be impossible to reconstruct the stream of random numbers prior to the revelation
  - additionally, if there is an entropy input while running, it should be infeasible to use knowledge of the input's state to predict future conditions of the CSPRNG state.

CSPRGN: statistical definition

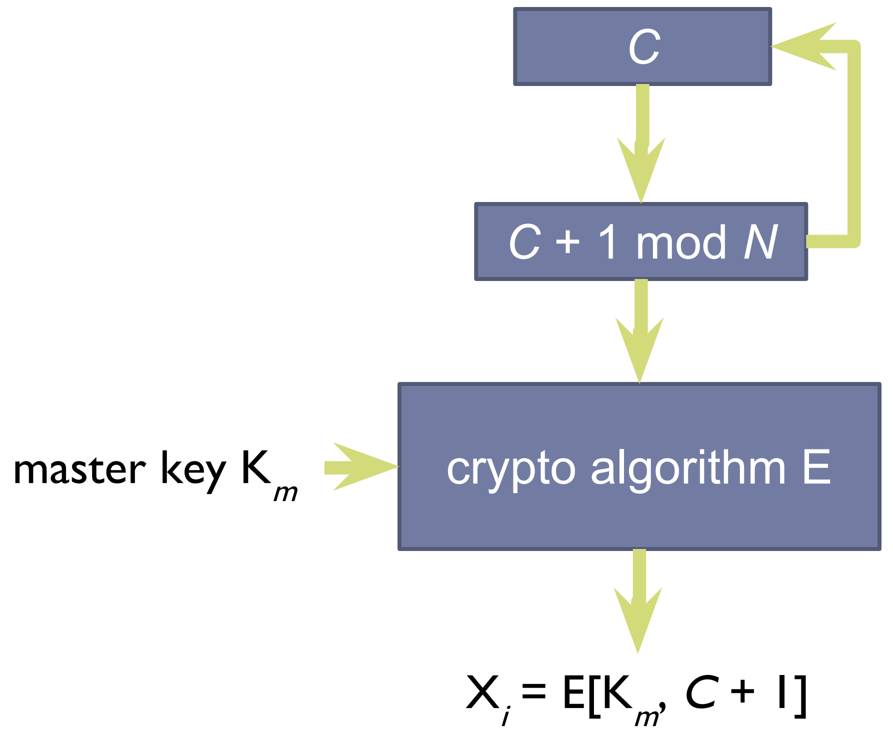
no polynomial time algorithm can distinguish with probability > 0.5 a truly random sequence from a pseudo-random sequence of the generator

open problem

is there any way to distinguish the output of a high-quality PRNG from a truly random sequence without knowing the algorithm(s) used and the state with which it was initialized?

generator "cipher of a counter"

- cyclic cryptography: use counter + cipher
  - Meyer and Matyas, 1982



RSA based generator

- prime numbers p, q
- $n = p \cdot q$
- integer e s.t. GCD(e, $\phi(n)$) = GCD(e, $(p-1)\cdot(q-1)$) = 1
- z = seed
- loop

```
    $z_i = (z_{i-1})^e \bmod n$
    $i = i + 1$
```

output: least significant bit of z_i

ANSI X9.17

- strong generator, using 3DES (triple DES, EDE)
- input: seed s of 64-bit, integer m, triple DES key, D (encoding of date and time)
- output: $x_1, x_2, \ldots, x_m$ sequence of m strings (64 bits in the whole)

```
$Y = 3DES(D)$
for $i = 1$ to $m$ do
    $x_i = 3DES(Y \oplus s)$
    $s = 3DES(x_i \oplus Y)$
return $x_1, x_2, \ldots, x_m$
```

Blum Blum Shub (CSPRNG)

- choose p, q big prime s.t. $p \equiv q \equiv 3 \pmod 4$
- $n = p \cdot q$
- randomly choose s s.t. GCD(s, n) = 1
- output the sequence of bits B_i

```
$X_0 = s^2 \bmod n$
for $i = 1$ to $\infty$
    $X_i = (X_{i-1})^2 \bmod n$
    $B_i = X_i \bmod 2$
    return $B_i$
```

more on randomness

- subroutines generally provided in programming languages for "random numbers" are not designed to be unguessable, but just designed merely to pass statistical tests
- hash together all the sources of randomness you can find, e.g., timing of keystrokes, disk seek times, count of packet arrivals, etc.

- for more reading about randomness, see RFC 1750.

RFC 1750 (1994)

Randomness Recommendations for Security

Security systems today are built on increasingly strong cryptographic algorithms that foil pattern analysis attempts. However, the security of these systems is dependent on generating secret quantities for passwords, cryptographic keys, and similar quantities. The use of pseudo-random processes to generate secret quantities can result in pseudo-security. The sophisticated attacker of these security systems may find it easier to reproduce the environment that produced the secret quantities, searching the resulting small set of possibilities, than to locate the quantities in the whole of the number space.

Choosing random quantities to foil a resourceful and motivated adversary is surprisingly difficult. This paper points out many pitfalls in using traditional pseudo-random number generation techniques for choosing such quantities. It recommends the use of truly random hardware techniques and shows that the existing hardware on many systems can be used for this purpose. It provides suggestions to ameliorate the problem when a hardware solution is not available. And it gives examples of how large such quantities need to be for some particular applications.

Exercise

Consider the following generator based on

- hash function H
- random initial seed *s*
  - y = *s*
  - for i = 1 to n do
    - y = H(y)

why it is not good?