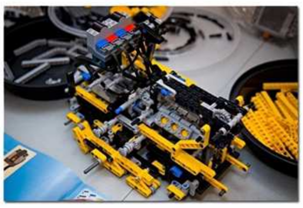
Random number generation

Security Protocols (bmevihim132)

Dr. Levente Buttyán
associate professor
BME Hálózati Rendszerek és Szolgáltatások Tanszék
Lab of Cryptography and System Security (CrySyS)
buttyan@hit.bme.hu, buttyan@crysys.hu

Outline

- motivations and definitions
  - attacks on an early version of the Netscape PRNG
- true random sources and entropy estimation
- cryptographic pseudo-random number generators (PRNGs)
  - general structure
  - attacker models
  - attacks on known PRNGs
  - the Yarrow-160 PRNG



Motivation

- random numbers (bits) are needed for various purposes, including for generating cryptographic keys (both symmetric and asymmetric) and other cryptographic parameters (e.g., unpredictable IVs, nonces, blinding parameters, etc.)
- random number generators used for simulation purposes are not good for cryptographic purposes
  - example: $s_{i+1} = (a \cdot s_i + b) \bmod n$
    - has nice statistical properties
    - but it is predictable
- weakly designed random number generators can easily destroy security even if very strong cryptographic primitives (ciphers, MACs, etc.) are used
  - eg., early version of Netscape PRNG (to be used for SSL)

Early version of Netscape's PRNG

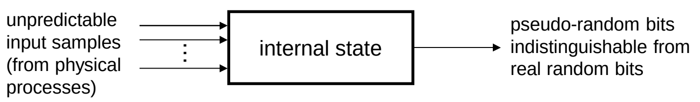
```
RNG_CreateContext()
      (seconds, microseconds) = time of day;
      pid = process ID; ppid = parent process ID;
      a = mklcpr(microseconds);
      b = mklcpr(pid + seconds + (ppid << 12) );
      seed = MD5(a, b);

mklcpr(x)
      return((0xDEECE66D*x + 0x2BBB62DC) >> 1)

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

create_key()
      RNG_CreateContext();
      RNG_GenerateRandomBytes(); RNG_GenerateRandomBytes();
      challenge = RNG_GenerateRandomBytes();
      secret_key =  RNG_GenerateRandomBytes();
```

Attacking the Netscape PRNG

- if an attacker has an account on the UNIX machine running the browser
  - *ps* command lists running processes → attacker learns pid, ppid
  - the attacker can guess the time of day with seconds precision
  - only unknown is the value of microseconds → ~2^{20} possibilities
  - each possibility can be tested easily against the challenge sent in clear within SSL
- if the attacker has no account on the machine running the browser
  - a has 20 bits of randomness, b has 27 bits of randomness → seed has 47 bits of randomness (compared to 128 bit advertised security)
  - ppid is often 1, or a bit smaller than pid
  - sendmail generates message IDs from its pid
    - send mail to an unknown user on the attacked machine
    - mail will bounce back with a message ID generated by sendmail
    - attacker learns the last process ID generated on the attacked machine
    - this may reduce possibilities for pid

Definitions

- a **random number** is a number that cannot be predicted by an observer before it is generated
  - if the number is generated within the range [0, N-1], then its value cannot be predicted with any better probability than 1/N
  - the above is true even if the observer is given all previously generated numbers
- a cryptographic **pseudo-random number generator (PRNG)** is a mechanism that processes somewhat unpredictable inputs and generates pseudo-random outputs
  - if designed, implemented, and used properly, then even an adversary with enormous computational power should not be able to distinguish the PRNG output from a real random sequence



Harvesting true random bits

- gathering bits unknown to and unguessable by the adversary
- possible sources:
  - keystroke timings
  - mouse movement
  - disc access time
  - noisy diodes or noisy resistors (quantum effects)
  - /dev/random
    - a UNIX device available under some systems which gathers entropy from system tables and events not available to any user
    - even if the adversary happens to be running a process on the machine, the bits provided by /dev/random are still secret
- collected bits are not necessarily all independent, the adversary might even know entire subsequences of the bits
- what is important is that the harvested bits contain information (entropy) which is unavailable to the adversary

Entropy estimation

- determining how many unguessable bits were harvested
- relevant concepts:
  - entropy: $$H = -\sum_x p_x \log_2(p_x)$$
    where x is a possible value in a stream of values and p_x is its probability of occurrence (from an infinite population of x values not just a finite sample)
  - entropy per source bit: $$J = \frac{H}{|x|}$$
    where |x| is the size of the symbol x in bits
  - absolute entropy: minimum entropy regardless of the symbol size: $$E = \min_{1 \le |x| < \infty} J$$

Exercises for entropy estimation

- exercise 1:
  - consider a source that repeatedly outputs 00 and 11 (2 bits/round) with equal probability
  - compute H and J when
    - x is 1 bit long (i.e. x is in {0, 1})
    - x is 2 bits long (i.e., x is in {00, 01, 10, 11})
    - x is 3 bits long
    - x is n bits long
  - what is the value of E for this source?

- exercise 2:
  - what is the value of E for a source that produces a periodic sequence?

Estimating E in practice

- determine compression ratio achieved for the harvested bits by the best available compression algorithm

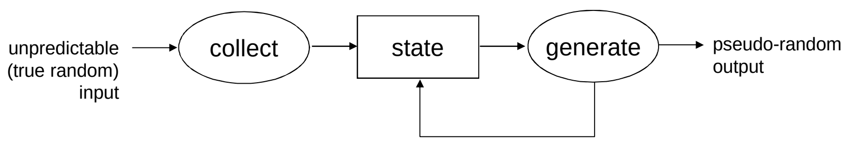
- this must be further reduced with the fraction of bits that an adversary might have acquired by guessing, measurement, or creating some bias in the generator process
  - e.g., if one uses the system date and time as a source of random bits, then one can expect the adversary to know the date, and to probably know the hour, and maybe the minutes
  - e.g., if one uses a mouse drawn signature as an entropy source, then only the noisy deviations from the usual signature count as entropy, as the adversary may know the usual signature

Reduction to independent bits

- compute a hash of the harvested bits to reduce them to independent random bits
  - the hash function needs to have each output bit functionally dependent on all input bits and functionally independent of all other output bits
  - in practice, cryptographic hash functions, such as SHA will do
- if the output size of the hash function is n, then feed it with at least n/E harvested input bits (and not much more)

PRNGs

- often, one needs more random bits than the available sources of entropy can provide → one needs a PRNG that produces pseudo-random numbers (bits) from a certain amount of true randomness (seed)
  - computationally limited adversaries will not be able to distinguish this pseudo-random sequence from a truly random sequence
  - if the PRNG is well-designed, then computationally limited adversaries will not be able to predict the PRNG's output
- general structure:



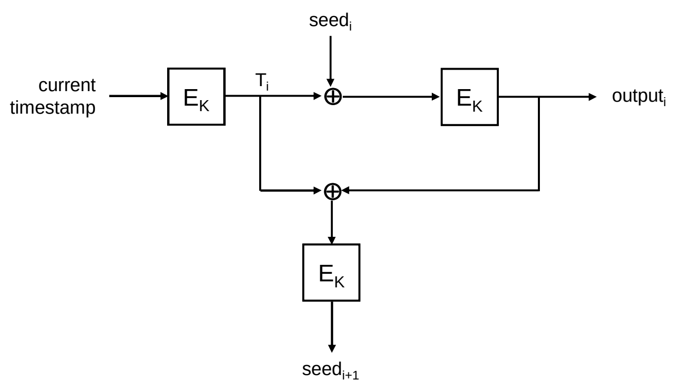
Classification of attacks

- various ways to compromise the PRNG's state
  - cryptanalytic attacks
    - between receiving input samples the PRNG works as a stream cipher
    - a cryptographic weakness in this stream cipher might be exploited to recover its internal state
  - input based attacks
    - known-input attacks: an attacker is able to observe (some of) the PRNG inputs
    - chosen-input attacks: an attacker is able to control (some of) the PRNG inputs
  - implementation attacks
    - mishandling of seed files
    - side-channel attacks
      - additional information about the actual implementation of the PRNG may be exploited
      - e.g., measuring the time needed to produce a new output may leak information about the current state of the PRNG (timing attacks)
- various ways to extend state compromise
  - iterative guessing attacks
    - figure out PRNG outputs produced after the state compromise
  - backtracking
    - figure out PRNG outputs produced before the state compromise

ANSI X9.17

state: K, $seed_i$
output generation:

$T_i = E_K(\text{current timestamp})$
$output_i = E_K(T_i \oplus seed_i)$
$seed_{i+1} = E_K(T_i \oplus output_i)$



Attacks on X9.17

- cryptanalytic attacks
  - it seems that they require to break the block cipher E
  - however, this has never been proven formally
- weaknesses leading to state compromise extensions
  - part of the state (K) never changes
    → if K is compromised, then the PRNG can never fully recover
  - $seed_{i+1}$ depends on $seed_i$ only via $output_i$
    → if K is known from a previous state compromise and $output_i$ is observable, then finding $seed_{i+1}$ is not so difficult (timestamps can usually be assumed to have only 10-20 bits of entropy)

Attacks on X9.17

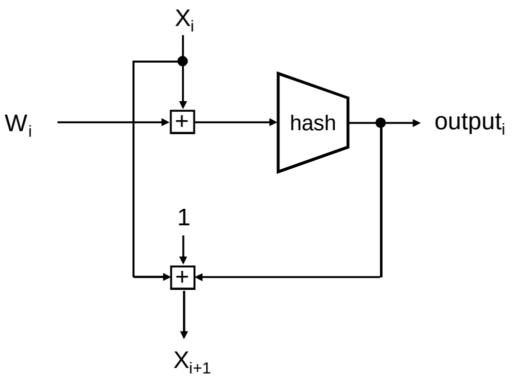
- iterative guessing attack
  - if an attacker knows K and $seed_i$ and sees (some public function f of) $output_i$, then he can determine $seed_{i+1}$ easily
    - let $f(output_i) = v$
    - try all possible values t for T_i, and form a list of values $v_t = f(E_K(t \oplus seed_i))$
    - select t* such that $v_{t^*} = v$
    - $seed_{i+1} = E_K(t^* \oplus E_K(t^* \oplus seed_i))$
- backtracking
  - if an attacker knows K and $seed_{i+1}$ and sees (some public function f of) $output_i$, then he can determine $output_i$ and $seed_i$ easily (EXERCISE)
- timer entropy issues
  - if larger amount of random bytes are needed (e.g., RSA key pair generation), then the PRNG is called repeatedly within a very short time

  → consecutive T_i values have much less entropy than 10-20 bits

DSA PRNG

state: X_i
optional input: W_i ($W_i = 0$ if not supplied)
output generation:

$$output_i = hash((W_i + X_i) \bmod 2^{160})$$
$$X_{i+1} = (X_i + output_i + 1) \bmod 2^{160}$$



Attacks on the DSA PRNG

- cryptanalytic attacks
  - if the hash function is good, then the PRNG output is hard to be distinguished from a real random sequence
  - no formal proof

- input based attacks
  - assume the attacker can control W_i
  - setting $W_i = (W_{i-1} - output_{i-1} - 1) \bmod 2^{160}$ will force the PRNG to repeat its output

$$\begin{aligned} output_i &= hash((W_i + X_i) \bmod 2^{160}) = \\ &= hash(((W_{i-1} - output_{i-1} - 1) + (X_{i-1} + output_{i-1} + 1)) \bmod 2^{160}) = \\ &= hash((W_{i-1} + X_{i-1}) \bmod 2^{160}) = \\ &= output_{i-1} \end{aligned}$$

  - this works only if input samples are sent directly into the PRNG
    - in practice, they are often hashed before sent in

Attacks on the DSA PRNG

- a weakness that may make state compromise extensions easier
  - X_{i+1} depends on W_i only via $output_i$
    → if an attacker compromised X_i and can observe $output_i$, then he knows X_{i+1} no matter how much entropy has been fed into the PRNG by W_i
- iterative guessing attack
  - if an attacker knows X_i and observes a public function f of $output_i$, then he can find X_{i+1}
    - let $f(output_i) = v$
    - assume that W_i has only 20 bits of entropy (e.g., it is obtained from a timestamp of microsecond precision)
    - the attacker can try all possible values w for W_i, and compute $v_w = f(hash((w + X_i) \bmod 2^{160}))$
    - let w* be the value such that $v = v_{w^*}$
    - $X_{i+1} = (X_i + hash((w^* + X_i) \bmod 2^{160}) + 1) \bmod 2^{160}$
- filling the gaps
  - if an attacker knows X_i and X_{i+2}, and observes $output_{i+1}$, then he can compute $output_i$ as

    $output_i$ = ??? (EXERCISE)

Some guidelines for using PRNGs

- use a hash function at the output to protect the PRNG from direct cryptanalytic attacks
- hash all inputs together with a counter or timestamp before feeding into the PRNG to make chosen-input attacks harder
- pay special attention to PRNG starting points and seed files to make it harder to compromise the PRNG state
- occasionally generate a new starting state and restart the PRNG to limit the scope of state compromise extensions

The Yarrow-160 PRNG

- design philosophy
  - accumulate entropy from as many different sources as possible
  - reseed (re-generate state) when enough entropy has been collected (this puts the PRNG in an unguessable state at each reseed)
  - between reseeds, use strong crypto algorithms to generate outputs from the internal state (like a stream cipher)
- four major components
  - entropy accumulator
    - collects samples from entropy sources into two entropy pools (slow and fast pool)
  - reseed control
    - determines when a reseed should be performed
  - reseed mechanism
    - reseeds the key with new entropy from the pools
  - generation mechanism
    - generates PRNG output from the state

Entropy accumulator

- inputs from each source are fed alternately into two entropy pools
  - fast pool
    - provides frequent reseeds
    - ensures that state compromises has as short a duration as possible
  - slow pool
    - rare reseeds
    - entropy is estimated conservatively
    - rationale: even if entropy estimation of the fast pool is inaccurate, the PRNG still eventually gets a secure reseed from the slow pool
- entropy estimation
  - entropy of each sample is measured in three ways:
    - a: programmer supplies an estimate for the entropy source
    - b: a statistical estimator is used to estimate the entropy of the sample
    - c: length of the sample multiplied by ½
  - entropy estimate of the sample is min(a, b, c)
  - entropy contribution of a source is the sum of entropy estimates of all samples collected so far from that source
  - entropy contribution of each source is maintained separately

Reseed control

- periodic reseed
  - the fast pool is used to reseed when any of the sources reaches an estimated entropy contribution of 100 bits
  - the slow pool is used to reseed when at least two sources reach an estimated entropy contribution of 160 bits
- explicit reseed
  - an application may explicitly ask for a reseed operation (from both pools)
  - should be used only when a high-valued random secret is to be generated

Reseed mechanism

- reseed from the fast pool (h is SHA1, E is 3DES):

  v_0 := h(fast pool)

  $v_i := h(v_{i-1} \mid v_0 \mid i)$ for $i = 1, 2, \ldots, P_t$

  $K := h'(h(v_{P_t} \mid K), k)$

  $C := E_K(0)$

  where h' is a "size adaptor"

  $h'(m, k)$ = first k bits of $s_0 \mid s_1 \mid s_2 \mid \ldots$

  $s_0 = m$

  $s_i = h(s_0 \mid \ldots \mid s_{i-1})$ $i = 1, 2, \ldots$

  \+ reset all entropy estimates to 0

  \+ clear the memory of all intermediate values

- reseed from the slow pool:
  - feed h(slow pool) into fast pool
  - reseed from fast pool as described above

Reseed mechanism

- observations
  - new value of K directly depends on previous value of K and current pool content (pool → v_0 → v_{Pt})
    - if an attacker has some knowledge of the previous value of K, but does not know most of the pool content, then he cannot guess the new K
    - if an attacker does not know the previous value of K, but observed many inputs of the pool, then he still cannot guess the new K
  - execution time depends on security parameter P_t
    - this makes the time needed for iterative guessing attacks longer

Generation mechanism

- algorithm (E is 3DES):

  C := (C+1) mod 2^n // n is the block size of E

  R := E_K(C)

  output: R

- generator gate
  - after P_g output has been generated, a new key is generated

    K := next k bits of PRNG output
  - P_g is a security parameter currently set to 10
  - rationale: if a key is compromised, then only 10 previous output can be computed by the attacker (prevention of backtracking attacks)

Protecting the entropy pool

- the pool may be swapped into swap files and stored on disk
  - several operating systems allow to lock pages into memory
    - mlock() (UNIX), VirtualLock() (Windows), HoldMemory() (Macintosh)
  - memory mapped files can be used as private swap files
    - the files should have the strictest possible access permissions
    - file buffering should be disabled to avoid that the buffer is swapped
- allocated memory blocks can be scanned through by other processes
  - entropy pool is often allocated at the beginning when the security subsystem is started → pool is often at the head of allocated memory blocks
  - the pool can be embedded in a larger allocated memory block
  - its location can be changed periodically (by allocating new space and moving the pool) in the background
  - this background process can also be used to prevent the pool from being swapped (touched pages are kept in memory with higher probability)

Summary

- random numbers for cryptographic purposes need special attention
  - simple congruential generators are predictable
  - naïve design will not do (cf. early Netscape PRNG)
- random sources and entropy estimation
- cryptographic pseudo-random number generators (PRNGs)
  - attacker models
  - some standardized PRNGs have weaknesses
    - e.g., ANSI X9.17, DSA PRNG, RSAREF 2.0, …
  - vulnerable PRNGs can be made stronger by adding some simple extensions (e.g., hash all inputs before sending into the PRNG)
  - the Yarow-160 PRNG
    - careful design that seems to resist various attacks

Recommended readings

- Ellison. P1363 Appendix E: Cryptographic Random Numbers,1995.
- Kelsey, Schneier, Wagner, Hall. Cryptographic attacks on PRNGs. Workshop on Fast Software Encryption, 1998.
- Kelsey, Schneier, Ferguson. Yarrow-160: Notes on the design and analysis of the Yarrow cryptographic PRNG.