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

Definition

\[
\text{unpredictable input samples} \quad \longleftrightarrow \quad \text{internal state} \quad \longleftrightarrow \quad \text{pseudo-random bits indistinguishable from real random bits}
\]
Motivation

- sources of true randomness may be available ...
  - keystroke timing
  - mouse movement
  - disc access time
  - network usage statistics
  - ...

- ... but the amount of random bits obtained per time unit or available at a given point in time may not be sufficient

- random number generators used for simulation purposes are not good for cryptographic purposes
  - example: \( s_{i+1} = (a \cdot s_i + b) \mod n \)
    - has nice statistical properties
    - but it is predictable

- weakly designed PRNGs can easily destroy security even if very strong cryptographic primitives (ciphers, MACs, etc.) are used
  - example: early version of Netscape PRNG (to be used for SSL)

---

Early version of Netscape’s PRNG

```c
RNG_CreateContext()
(\text{seconds, microseconds}) = \text{time of day};
pid = \text{process ID}; ppid = \text{parent process ID};
a = \text{mklcpr(\text{microseconds})};
b = \text{mklcpr(pid + seconds + (ppid \ll 12))};
seed = \text{MD5(a, b)};

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

RNG_GenerateRandomBytes()
\text{x = MD5(seed)};
\text{seed = seed+1};
\text{return x};

create_key()
RNG_CreateContext();
RNG_CreateRandomBytes(); RNG_CreateRandomBytes();
\text{challenge = RNG_CreateRandomBytes()};
\text{secret_key = RNG_CreateRandomBytes()};
```
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 → \(2^{22}\) 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`

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
  - side-channel attacks
    - additional information about the actual implementation of the PRNG may be exploited
    - example: measuring the time needed to produce a new output may leak information about the current state of the PRNG (timing attacks)
      ```
      x = MD5(seed);
      seed = seed+1;   // increment needs m+1 byte additions if the last m bytes are all 0xFF
      return x;       // long output time  → last couple of bytes of seed are 0x00
      ```
  - 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
      - typically applicable against smart cards
  - mishandling of seed files
Classification of attacks

- in practice, it is prudent to assume that occasional compromises of the state may happen
- various ways to exploit compromised states
  - permanent compromise attacks
    - given: state at time $t_0$
    - find: all future (or past) states
  - iterative guessing attacks
    - given: state at time $t_0$, outputs in $[t_0, t_1]$
    - find: state at time $t_1$
  - backtracking attacks
    - given: state at time $t_0$
    - find: outputs before $t_0$
  - meet-in-the-middle attacks
    - given: state at time $t_0$ and $t_2 > t_0$
    - find: state at time $t_1$, where $t_0 < t_1 < t_2$

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

- input based attacks
  - assume that an attacker can freeze the clock ($T_i = T$ for all $i$)
  - $\text{output}_{i+1} = E_k(T \oplus \text{seed}_{i+1}) = E_k(T \oplus E_k(T \oplus \text{output}_i)) = E'_k(\text{output}_i)$
  - for a good cipher $E$, we expect a repeating value in the above sequence after $\approx 2^{n-1}$ steps, where $n$ is the block size of $E$
  - in a sequence of true $n$-bit random values, a collision is expected after $\approx 2^{n/2}$ steps (birthday paradox)
  - the attacker can distinguish the output of X9.17 from a sequence of true random numbers given that he can observe sufficiently many ($\approx 2^{n/2}$) outputs
    - not practically important
    - certificational weakness

- weaknesses leading to state compromise extensions
  - part of the state ($K$) never changes
    - if $K$ is compromised, then the PRNG can never fully recover
  - $\text{seed}_{i+1}$ depends on $\text{seed}_i$ only via $\text{output}_i$
    - if $K$ is known from a previous state compromise and $\text{output}_i$ is observable, then finding $\text{seed}_{i+1}$ is not so difficult (timestamps can usually be assumed to have only 10-20 bits of entropy)

- deriving the seed from two consecutive outputs (and $K$)
  $\text{seed}_{i+1} = E_k(T_i \oplus \text{output}_i)$ \hspace{1cm} (1)
  $\text{seed}_{i+1} = D_k(\text{output}_{i+1}) \oplus T_{i+1}$ \hspace{1cm} (2)
  - assume that timestamps has 10 bits of entropy
  - try all values for $T_i$, and form a sorted list of possible values for $\text{seed}_{i+1}$ using (1)
  - try all values for $T_{i+1}$, and form another sorted list of possible values for $\text{seed}_{i+1}$ using (2)
  - the correct $\text{seed}_{i+1}$ value is the one that appears on both lists
    (expected number of matching pairs is $\approx 1-2^{20-n}$)
**Attacks on X9.17**

- **iterative guessing attack**
  - if an attacker knows K and seed, and sees (some public function $f$ of) output, 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, and sees (some public function $f$ of) output, then he can determine output, and seed, 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
  - $\rightarrow$ 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:
- $output_i = \text{hash}((W_i + X_i) \mod 2^{160})$
- $X_{i+1} = (X_i + output_i + 1) \mod 2^{160}$
Attacks on the DSA PRNG

- Cryptanalytic attacks
  - If the hash function is good, then the PRNG output seems to be hard to distinguish from a real random sequence.
  - No formal proof.

- Input based attacks
  - Assume the attacker can control $W_i$.
  - Setting $W_i = (W_{i-1} - \text{output}_{i-1} - 1) \mod 2^{160}$ will force the PRNG to repeat its output.
    
    $$\text{output}_i = \text{hash}((W_i + X_i) \mod 2^{160}) = \text{hash}(((W_{i-1} - \text{output}_{i-1} - 1) + (X_{i-1} + \text{output}_{i-1} + 1)) \mod 2^{160}) = \text{hash}((W_{i-1} + X_{i-1}) \mod 2^{160}) = \text{output}_{i-1}$$
  - This works only if input samples are sent directly into the PRNG.
    - In practice, they are often hashed before sent in.

- 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.
    - The attacker can try all possible values $w$ for $W_i$, and compute
      $$v_w = f(\text{hash}((w + X_i) \mod 2^{160}))$$
    - Let $w*$ be the value such that $v = v_{w*}$.
    - $X_{i+1} = (X_i + \text{hash}((w* + X_i) \mod 2^{160}) + 1) \mod 2^{160}$

- 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
    $$\text{output}_i = (X_{i+2} - X_i - 2 - \text{output}_{i+1}) \mod 2^{160}$$
Strengthening the DSA PRNG

- all inputs should be hashed together before feeding them into the PRNG (to make input based attacks harder)
- $X_{i+1}$ should depend on $W_i$ directly and not via the output
  - example: $X_{i+1} = X_i + \text{hash}($output$ + W_i)$

Guidelines for using vulnerable 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 the key (state) only 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 key (like a stream cipher)

- four major components
  - entropy accumulator
    - collects samples from entropy sources into two entropy pools (slow and fast pool)
  - reseed mechanism
    - periodically reseeds the key with new entropy from the pools
  - reseed control
    - determines when a reseed should be performed
  - generation mechanism
    - generates PRNG output from the key (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 very 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 \( \frac{1}{2} \)
  - 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 reaches 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):
  \[ \begin{align*}
  v_0 & := h(\text{fast pool}) \\
  v_i & := h(v_{i-1} | v_0 | i) \quad \text{for } i = 1, 2, \ldots, P_t \\
  K & := h'(h(v_{P_t} | K), k) \\
  C & := E_k(0)
  \end{align*} \]
  where \( h' \) is a "size adaptor"
  \( h'(m, k) = \text{first } k \text{ bit of } s_0 \mid s_1 \mid s_2 \mid \ldots \)
  \( s_0 = m \)
  \( s_i = h(s_0 \mid \ldots \mid s_{i-1}) \quad i = 1, 2, \ldots \)
  reset all entropy estimates to 0
  wipe the memory of all intermediate values

- reseed from the slow pool:
  - feed \( h(\text{slow pool}) \) 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 ($\text{pool} \to v_0 \to v_{n_t}$)
    - 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):
  \[
  C := (C+1) \mod 2^n \quad \text{// $n$ is the block size of $E$}
  R := E_K(C)
  \]
  output: $R$

- generator gate
  - after $P_g$ output has been generated, a new key is generated
    $K := \text{next } k \text{ bits of PRNG output}$
  - $P_g$ 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 can 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

- PRNGs for cryptographic purposes need special attention
  - Simple congruential generators are predictable
  - Naïve PRNG design will not do (cf. early Netscape PRNG)
- Widely used cryptographic PRNGs may have weaknesses too
  - ANSI X9.17
  - DSA PRNG
  - RSAREF 2.0
  - ...
- Some guidelines for using vulnerable PRNGs
- Design of Yarow-160
  - Careful design that seems to resist various attacks
- Protecting the entropy pools
Recommended readings

- Kelsey, Schneier, Ferguson. Yarrow-160: Notes on the design and analysis of the Yarrow cryptographic PRNG.