Cryptanalysis of the Windows Random Number Generator

Masaryk University in Brno
Faculty of Informatics

Jan Krhovják

Outline

- Basic information
- Analysis of the Windows PRNG
- Attacks on forward security
- Attacks on backward security
- Usage of Windows PRNG
Introduction

- Pseudo-randomness
  - Output of PRNG looks random to outside observer

- Forward security
  - The attacker that know internal state of PRNG cannot learn anything about previous outputs

- Backward security (break-in recovery)
  - The attacker that know internal state of PRNG cannot learn anything about future outputs

- Attacker model
  - Attacker need a state of generator (application, buffer overflow)
    - No additional information from attacked system
  - Compromised state => all previous and future PRNG outputs
The structure of Windows PRNG (WRNG)

- Generator from Windows 2000 binary code
- Based on RC4 and variant of SHA-1 (different IVs)
  - Generates 20 bytes per iteration
    ```
    // output Len bytes to buffer
    while (Len > 0) {
        R := R ⊕ get_next_20_rc4_bytes()
        State := State ⊕ R
        T := SHA-1′(State)
        Buffer := Buffer | T
        // | denotes concatenation
        R[0..4] := T[0..4]
        // copy 5 least significant bytes
        State := State + R + 1
        Len := Len − 20
    }
    ```
- Main state is composed from registers $R$ and $State$
  - Not explicitly initialized => latest values in allocated memory
The function get_next_20_RC4_bytes

- Function keeps its own state
  - Eight instances of RC4
  - In each call function performs
    - Select 1 RC4 state (round-robin)
    - Use it to generate 20 byte output
  - Refresh after RC4 instance generates 16 Kbytes of data

- Initializing and refreshing each instance of RC4
  - Entropy gathered from system
  - Collection of 3584 bytes of data
    - Hashed to produce 80-byte digest
  - Used for encryption of seed
    - Accessible in clear-text from registry
  - RC4 encryption of data from driver KSecDD => final RC4 key

- Entropy sources

<table>
<thead>
<tr>
<th>Source</th>
<th>Size in bytes requested</th>
</tr>
</thead>
<tbody>
<tr>
<td>CircularHash</td>
<td>256</td>
</tr>
<tr>
<td>KSecDD</td>
<td>256</td>
</tr>
<tr>
<td>GetCurrentProcessID()</td>
<td>8</td>
</tr>
<tr>
<td>GetCurrentThreadId()</td>
<td>8</td>
</tr>
<tr>
<td>GetTickCount()</td>
<td>8</td>
</tr>
<tr>
<td>GetLocalTime()</td>
<td>16</td>
</tr>
<tr>
<td>QueryPerformanceCounter()</td>
<td>24</td>
</tr>
<tr>
<td>GlobalMemoryStatus()</td>
<td>16</td>
</tr>
<tr>
<td>GetDiskFreeSpace()</td>
<td>40</td>
</tr>
<tr>
<td>GetComputerName()</td>
<td>16</td>
</tr>
<tr>
<td>GetUserName()</td>
<td>257</td>
</tr>
<tr>
<td>GetCursorPos()</td>
<td>8</td>
</tr>
<tr>
<td>GetMessageTime()</td>
<td>16</td>
</tr>
<tr>
<td>NTQuerySystemInformation calls</td>
<td></td>
</tr>
<tr>
<td>ProcessorTimes</td>
<td>48</td>
</tr>
<tr>
<td>ProcessorStatistics</td>
<td>up to the remaining length</td>
</tr>
<tr>
<td>Performance</td>
<td>312</td>
</tr>
<tr>
<td>Exception</td>
<td>16</td>
</tr>
<tr>
<td>Lookaside</td>
<td>32</td>
</tr>
<tr>
<td>ProcessesAndThreads</td>
<td>up to the remaining length</td>
</tr>
</tbody>
</table>
Scope of WPRNG

- One WPRNG per process (& in user mode)
  - Different process have separate internal states
    - RC4 states (and other variables) are in DLL space
    - $R$ and $State$ are in stack
  - Different threads in one process
    - Share RC4 states
    - Have own stack (and copy of $R/State$)

- Impacts of scoping
  - Breaking one WRNG does not affect another WPRNG
  - Only one consumer per WRNG => long period between rekeys
Attack on backward security

- Suppose that an adversary knows internal state
- The next state and output are deterministic function of the actual state
- An adversary can compute next state and output until the next refresh of generator
Attacks on forward security

- **Instant attack**
  - Initial values, state $R$ and $State$, 8 RC4 registers are known
  - RC4 does not provide forward security
  - Given a current state we can compute previous states/outputs

- **Attacks with overhead $2^{40}$ and $2^{23}$**
  - State $R$ and $State$ are unknown $\Rightarrow$ more complex attacks
    - Based on relations between several values or operations
    - Typically R/S; addition / exclusive or operations
  - The latter attack on Pentium IV 2.80 GHz takes 19 seconds

- **Bad design of state updates (xor and +)**
  - $S^{t+1}=(S^txor R)+R'+1$: $R$ is almost identical to $R'$ (5 bytes replaced with output bytes) $\Rightarrow$ $S^{t+1}$ is strongly related to $S^{t+1}$
Interaction between OS and Generator I

- Frequency of entropy based rekeys of state
  - 1 process: 1 instance WRNG, 8 RC4 streams => refresh after each 128Kbytes of generated data
  - Between refreshes is generation deterministic

- Entropy based rekeys in Internet Explorer (sec. sensitive app.)
  - During SSL: 4 or more requests for 8, 16, 28 bytes of random data
  - Each SSL connection consumes approx. 100–200 bytes
  - IE asks for refresh only after handling 600–1200 SSL connections

- Initializing state $R$ and $State$ (not explicitly initialized)
  - First experiment: IE started after rebooting OS
    - Different mappings but correlated values
  - Second experiment: IE was restarted 20 times
    - Always same values
  - Third experiment: IE ran in 20 parallel sessions
    - In 19 cases correlated initial values (Hamming distance 10 or less)
Comparison to the Linux PRNG (LRNG)

**WRNG**
- User mode
  - App. can read state
- Reseeding timeout after 128KBytes of output
- Synchr. collecting of entropy
  - Entropy collected in periods
- Multiple runs of WRNG
- Attack on forward secrecy requires work $2^{64}$
- No blocking
  - No entropy measurements

**LRNG**
- Kernel mode
  - State is hidden to app.
- Reseeding timeout in every iteration
- Asynchr. collecting of entropy
  - Entropy event $=>$ pool update
- Single run of LRNG
- Attack on forward secrecy requires work $2^{23}$
- Possible blocking (DoS)
  - Entropy counter
Conclusions

- Security through obscurity (or even implementation complexity) do not work
  - Successful attacks are only a question of time
  - Attack on WRNG but also to (complex) open-source LRNG
- WRNG depends on RC4 that do not provide forward security
  - The recommendation is at least to replace this function
  - Better approach: replace whole WRNG and use, for example, by rigorously analyzed Barak-Halevi construction
    - If building blocks are secure then B-H construction provably preservers both forward and backward security
- WRNG also should rekey its state more often
  - Forced rekeys in some time intervals