Analysis of the Linux random number generator

(Presentation based on article of Z. Gutterman, B. Pinkas, and T. Reinman)

Jan Krhovják

Faculty of Informatics, Masaryk University
Article outline

- Random number generator in Linux – unique combination of TRNG and PRNG
  - A part of a Linux kernel
  - About 2500 lines of code
    - Poorly documented
    - Hundreds of (undocumented) patches
- Reverse engineering used for generator analysis
  - One bug in code itself
  - The problem with forward security
  - Several other design flaws
- Fundamentals of random number generation
  - Terminology issue (jargon in this field):
    term ”entropy” instead of ”data with entropy”
Article outline

- Random number generator in Linux – unique combination of TRNG and PRNG
  - A part of a Linux kernel
  - About 2500 lines of code
    - Poorly documented
    - Hundreds of (undocumented) patches

- Reverse engineering used for generator analysis
  - One bug in code itself
  - The problem with forward security
  - Several other design flaws

- Fundamentals of random number generation
  - Terminology issue (jargon in this field):
    term "entropy" instead of "data with entropy"
Article outline

- Random number generator in Linux – unique combination of TRNG and PRNG
  - A part of a Linux kernel
  - About 2500 lines of code
    - Poorly documented
    - Hundreds of (undocumented) patches

- Reverse engineering used for generator analysis
  - One bug in code itself
  - The problem with forward security
  - Several other design flaws

- Fundamentals of random number generation
  - Terminology issue (jargon in this field):
    term ”entropy” instead of ”data with entropy”
Random number generation

- Truly random data (samples) generated by TRNG
  - Hardware-based TRNG
    - Exact timing of keystrokes or exact movements of mouse
  - Software-based TRNG
    - Process, network, or I/O completion statistics
  - Difficulty of collecting sufficient amount truly random data
    $\Rightarrow$ the need of pseudo-random data

- Pseudorandom data generated by PRNG
  - PRNG is deterministic finite state machine
    $\Rightarrow$ at any point of time it is in a certain internal state
    - PRNG state is secret (PRNG output must be unpredictable)
    - PRNG (whole) state is repeatedly updated (PRNG must produce different outputs)

- The problem of state compromitting
  $\Rightarrow$ need of recovering from state compromise
  $\Rightarrow$ periodic state refreshing
  $\Rightarrow$ pooling
Random number generation

- Truly random data (samples) generated by TRNG
  - Hardware-based TRNG
    - Exact timing of keystrokes or exact movements of mouse
  - Software-based TRNG
    - Process, network, or I/O completion statistics
  - Difficulty of collecting sufficient amount truly random data
    $\Rightarrow$ the need of pseudo-random data

- Pseudorandom data generated by PRNG
  - PRNG is deterministic finite state machine $\Rightarrow$
    at any point of time it is in a certain internal state
    - PRNG state is secret (PRNG output must be unpredictable)
    - PRNG (whole) state is repeatedly updated (PRNG must produce different outputs)

- The problem of state compromitting $\Rightarrow$ need of recovering from state compromise $\Rightarrow$ periodic state refreshing $\Rightarrow$ pooling
Random number generation

- Truly random data (samples) generated by TRNG
  - Hardware-based TRNG
    - Exact timing of keystrokes or exact movements of mouse
  - Software-based TRNG
    - Process, network, or I/O completion statistics
  - Difficulty of collecting sufficient amount truly random data
    $\Rightarrow$ the need of pseudo-random data

- Pseudorandom data generated by PRNG
  - PRNG is deterministic finite state machine
    $\Rightarrow$
    at any point of time it is in a certain internal state
    - PRNG state is secret (PRNG output must be unpredictable)
    - PRNG (whole) state is repeatedly updated (PRNG must produce different outputs)

- The problem of state compromitting
  $\Rightarrow$ need of recovering from state compromise
  $\Rightarrow$ periodic state refreshing
  $\Rightarrow$ pooling
Linux (pseudo)random number generator (LRNG)

- Access to the LRNG through two device drivers
  - /dev/random and /dev/urandom

- Both devices let users read pseudorandom bits
  - Difference – the level of security and resulting delay
  - Blocked /dev/random and non-blocked /dev/urandom

- Basic structure of the LRNG – three asynchronous components:
  - 1st translates system events into bits
  - 2nd adds these bits to the LFSR-based generator pool
  - 3rd applies three consecutive SHA-1 operations to generate the output (feedback also entered back into the pool)

- Each sample of "randomness" (from system events) collected as two 32-bit words
  - The first word: measures the time of the event
  - The second word: event value (usually encoding of pressed key, mouse movement, drive access time, interrupt)
Linux (pseudo)random number generator (LRNG)

- Access to the LRNG through two device drivers
  - /dev/random and /dev/urandom

- Both devices let users read pseudorandom bits
  - Difference – the level of security and resulting delay
    - Blocked /dev/random and non-blocked /dev/urandom

- Basic structure of the LRNG – three asynchronous components:
  - 1st translates system events into bits
  - 2nd adds these bits to the LFSR-based generator pool
  - 3rd applies three consecutive SHA-1 operations to generate the output (feedback also entered back into the pool)

- Each sample of ”randomness” (from system events) collected as two 32-bit words
  - The first word: measures the time of the event
  - The second word: event value (usually encoding of pressed key, mouse movement, drive access time, interrupt)
Linux (pseudo)random number generator (LRNG)

- Access to the LRNG through two device drivers
  - /dev/random and /dev/urandom

- Both devices let users read pseudorandom bits
  - Difference – the level of security and resulting delay
  - Blocked /dev/random and non-blocked /dev/urandom

- Basic structure of the LRNG – three asynchronous components:
  - 1\textsuperscript{st} translates system events into bits
  - 2\textsuperscript{nd} adds these bits to the LFSR-based generator pool
  - 3\textsuperscript{rd} applies three consecutive SHA-1 operations to generate the output (feedback also entered back into the pool)

- Each sample of ”randomness” (from system events) collected as two 32-bit words
  - The first word: measures the time of the event
  - The second word: event value (usually encoding of pressed key, mouse movement, drive access time, interrupt)
Linux (pseudo)random number generator (LRNG)

- Access to the LRNG through two device drivers
  - `/dev/random` and `/dev/urandom`

- Both devices let users read pseudorandom bits
  - Difference – the level of security and resulting delay
    - Blocked `/dev/random` and non-blocked `/dev/urandom`

- Basic structure of the LRNG – three asynchronous components:
  - 1\textsuperscript{st} translates system events into bits
  - 2\textsuperscript{nd} adds these bits to the LFSR-based generator pool
  - 3\textsuperscript{rd} applies three consecutive SHA-1 operations to generate the output (feedback also entered back into the pool)

- Each sample of ”randomness” (from system events) collected as two 32-bit words
  - The first word: measures the time of the event
  - The second word: event value (usually encoding of pressed key, mouse movement, drive access time, interrupt)
Pools and counters

- Internal state kept in three entropy pools:
  - Primary (512 B), secondary (128 B), and urandom (128 B)
  - Entropy sources add data to the primary (or secondary) pool
  - Output generated from secondary/urandom pool

- Entropy extraction/transfer → feedback (hash of extracted bits)
- Each pool has its own entropy estimation counter
  - Important especially for secondary pool
Estimating the entropy amount

- Entropy of event is a function of its timing only
  - Type of event is not important
  - Let timing of event number \( n \) is \( t_n \). Define
    \[
    \delta_n = t_n - t_{n-1}; \quad \delta_n^2 = \delta_n - \delta_{n-1}; \quad \delta_n^3 = \delta_n^2 - \delta_{n-1}^2
    \]
    \( t_n, \delta_n, \delta_n^2, \delta_n^3 \) are each 32bit long
  - Amount of entropy added is defined as
    \[
    \log_2(\min(|\delta_n|, |\delta_n^2|, |\delta_n^3|)_{[19-30]}), \quad \text{where } S_{[19-30]} \text{ denotes bits } a \text{ to } b \text{ of } S
    \]
- Entropy counter updated only if estimation is positive
  - Pool is updated even if estimation is equal to 0
- Estimation is relevant only for OS sources
  - When user writes data to device – counter not incremented
- Extraction/transfer of \( n \) bits \( \rightarrow \) estimation is decremented by \( n \)
  - After transfer is counter in target pool incremented by \( n \)
Estimating the entropy amount

- Entropy of event is a function of its timing only
  - Type of event is not important
  - Let timing of event number \( n \) is \( t_n \). Define
    \[ \delta_n = t_n - t_{n-1}; \delta_n^2 = \delta_n - \delta_{n-1}; \delta_n^3 = \delta_n^2 - \delta_{n-1}^2 \]
    \( t_n, \delta_n, \delta_n^2, \delta_n^3 \) are each 32bit long
  - Amount of entropy added is defined as
    \[ \log_2(\min(|\delta_n|, |\delta_n^2|, |\delta_n^3|)_{[19-30]}) \]
    where \( S_{[19-30]} \) denotes bits \( a \) to \( b \) of \( S \)

- Entropy counter updated only if estimation is positive
  - Pool is updated even if estimation is equal to 0

- Estimation is relevant only for OS sources
  - When user writes data to device – counter not incremented

- Extraction/transfer of \( n \) bits \( \Rightarrow \) estimation is decremented by \( n \)
  - After transfer is counter in target pool incremented by \( n \)
Estimating the entropy amount

- Entropy of event is a function of its timing only
  - Type of event is not important
  - Let timing of event number \( n \) is \( t_n \). Define
    \[
    \delta_n = t_n - t_{n-1}; \quad \delta_n^2 = \delta_n - \delta_{n-1}; \quad \delta_n^3 = \delta_n^2 - \delta_{n-1}^2
    \]
    \( t_n, \delta_n, \delta_n^2, \delta_n^3 \) are each 32bit long
  - Amount of entropy added is defined as
    \[
    \log_2(\min(|\delta_n|, |\delta_n^2|, |\delta_n^3|)_{[19-30]}), \text{ where } S_{[19-30]} \text{ denotes bits } a \text{ to } b \text{ of } S
    \]

- Entropy counter updated only if estimation is positive
  - Pool is updated even if estimation is equal to 0

- Estimation is relevant only for OS sources
  - When user writes data to device – counter not incremented

- Extraction/transfer of \( n \) bits \( \Rightarrow \) estimation is decremented by \( n \)
  - After transfer is counter in target pool incremented by \( n \)
Estimating the entropy amount

- Entropy of event is a function of its timing only
  - Type of event is not important
  - Let timing of event number $n$ is $t_n$. Define
    $$\delta_n = t_n - t_{n-1}; \quad \delta_n^2 = \delta_n - \delta_{n-1}; \quad \delta_n^3 = \delta_n^2 - \delta_{n-1}^2$$
    $t_n, \delta_n, \delta_n^2, \delta_n^3$ are each 32bit long
  - Amount of entropy added is defined as
    $$\log_2(\min(|\delta_n|, |\delta_n^2|, |\delta_n^3|)_{[19\text{--}30]})$$
    where $S_{[19\text{--}30]}$ denotes bits $a$ to $b$ of $S$

- Entropy counter updated only if estimation is positive
  - Pool is updated even if estimation is equal to 0

- Estimation is relevant only for OS sources
  - When user writes data to device – counter not incremented

- Extraction/transfer of $n$ bits $\implies$ estimation is decremented by $n$
  - After transfer is counter in target pool incremented by $n$
Updating the pools

- Based on twisted generalized feedback shift register (TGFSR)
  - The main advantage is extended cycle/period length
    - The period of a TGFSR with a state of 128 words (on a 32-bit PC) can be $2^{128\times32} - 1$ steps
  - The implementation allows adding entropy in each iteration
    - Pools implemented as (indexed) arrays of 128 or 32 words
    - Adding entropy $\Rightarrow$ array index also updated

- Each pool is updated based on a primitive polynomial
  - Polynomial chosen according to the size of the pool
    - For primary pool: $x^{128} + x^{103} + x^{76} + x^{51} + x^{25} + x + 1$
    - For secondary/urandom pool: $x^{32} + x^{26} + x^{20} + x^{14} + x^{7} + x + 1$
  - Entropy addition can be viewed as reseeding in each iteration
    - Reseeding process changes the elementary properties of the TGFSR
    - The process is no longer linear function of initial state/seed
    - Long cycle/period can be no longer guaranteed :-(
Updating the pools

- Based on twisted generalized feedback shift register (TGFSR)
  - The main advantage is extended cycle/period length
    - The period of a TGFSR with a state of 128 words (on a 32-bit PC) can be $2^{128 \times 32} - 1$ steps
  - The implementation allows adding entropy in each iteration
    - Pools implemented as (indexed) arrays of 128 or 32 words
    - Adding entropy $\Rightarrow$ array index also updated

- Each pool is updated based on a primitive polynomial
  - Polynomial chosen according to the size of the pool
    - For primary pool: $x^{128} + x^{103} + x^{76} + x^{51} + x^{25} + x + 1$
    - For secondary/urandom pool: $x^{32} + x^{26} + x^{20} + x^{14} + x^{7} + x + 1$
  - Entropy addition can be viewed as reseeding in each iteration
    - Reseeding process changes the elementary properties of the TGFSR
    - The process in no longer linear function of initial state/seed
    - Long cycle/period can be no longer guaranteed :-(

Jan Krhovják (FI MU) Analysis of the Linux (P)RNG 20. 11. 2006 7 / 12
Extracting random bits

- Hashing the extracted bits, modifying the pools state, and decrementing the entropy estimate by the number of extracted bits
  - Process described for urandom or secondary pools (32 words long)
    - Decrementing entropy estimation & entropy refilling process omitted

- \((SHA-1)’\) uses as IVs 5 words of previous hash result
- Folding makes from 5 words (160 bits) 2.5 words (80 bits)
  - \(W_0, W_1, W_2, W_3, W_4\) yields \(W_0 \oplus W_3, W_1 \oplus W_4, W_2_{[0-15]} \oplus W_2_{[16-31]}\)
Forward security

**Definition:** An adversary which learns the internal state of the generator at a specific time cannot learn anything about previous outputs of the generator.

- Output computed after the state of pool is updated
  - Observation: with knowledge of state in time $t$ can be computed output in time $t - 1$

- Attack allows compute state in time $t - 1$, then in time $t - 2$, ...
  - Applicable when the pool entropy is not often updated
  - WLOG imagine XOR mod $2^{32} - 1$ instead addition over TGFSR
  - Generic attack with overhead $2^{96}$ (still impractical)
    - Only three 32bit values changed during extraction process
    - Much better then exhaustive search (overhead $2^{1024}$ for 32 word pool)
  - A more efficient attack with overhead $2^{64}$
    - Pool can be reversed for 18 of 32 index values (1,2,16,...,31)
    - For index 18, ..., 31 affected only words in upper half of pool
Forward security

- **Definition:** An adversary which learns the internal state of the generator at a specific time cannot learn anything about previous outputs of the generator.

- Output computed after the state of pool is updated
  - Observation: with knowledge of state in time $t$ can be computed output in time $t - 1$

- Attack allows compute state in time $t - 1$, then in time $t - 2$, ...
  - Applicable when the pool entropy is not often updated
  - WLOG imagine XOR mod $2^{32} - 1$ instead addition over TGFSR
  - Generic attack with overhead $2^{96}$ (still impractical)
    - Only three 32bit values changed during extraction process
    - Much better then exhaustive search (overhead $2^{1024}$ for 32 word pool)
  - A more efficient attack with overhead $2^{64}$
    - Pool can be reversed for 18 of 32 index values (1,2,16,...,31)
    - For index 18, ..., 31 affected only words in upper half of pool
Forward security

- **Definition:** An adversary which learns the internal state of the generator at a specific time cannot learn anything about previous outputs of the generator.

- Output computed after the state of pool is updated
  - Observation: with knowledge of state in time \( t \) can be computed output in time \( t - 1 \)

- Attack allows compute state in time \( t - 1 \), then in time \( t - 2 \), ...
  - Applicable when the pool entropy is not often updated
  - WLOG imagine XOR mod \( 2^{32} - 1 \) instead addition over TGFSR
  - Generic attack with overhead \( 2^{96} \) (still impractical)
    - Only three 32bit values changed during extraction process
    - Much better then exhaustive search (overhead \( 2^{1024} \) for 32 word pool)
  - A more efficient attack with overhead \( 2^{64} \)
    - Pool can be reversed for 18 of 32 index values (1,2,16,...,31)
    - For index 18, ..., 31 affected only words in upper half of pool
Security engineering

- No limits for reading from devices $\Rightarrow$ denial of service attacks
  - Local attacker: simply reads from `/dev/random` device
  - Remote attacker: can establish many TCP connections (TCP/SYN requires 128 bits of random data from urandom pool)
  - Solution: definition of quotas per user/group

- Guessable passwords (applicable on disk-less systems)
  - First user-operation in a computer system is user login
  - LRNG state might be a deterministic function of initial user password
  - Solution: keyboard entropy based on timing (not on typed values)

- An adversary can create noise that directly affects the LRNG output
  - Full primary pool $\Rightarrow$ entropy is added directly to secondary pool
  - Attacker can directly affect the generators output
  - Solution: always add entropy to primary pool

- The LRNG state reveals the previous LRNG output
  - Solution: switch order of operations (state update after LRNG output)
Security engineering

- No limits for reading from devices \(\Rightarrow\) denial of service attacks
  - Local attacker: simply reads from `/dev/random` device
  - Remote attacker: can establish many TCP connections
    (TCP/SYN requires 128 bits of random data from urandom pool)
  - Solution: definition of quotas per user/group

- Guessable passwords (applicable on disk-less systems)
  - First user-operation in a computer system is user login
  - LRNG state might be a deterministic function of initial user password
  - Solution: keyboard entropy based on timing (not on typed values)

- An adversary can create noise that directly affects the LRNG output
  - Full primary pool \(\Rightarrow\) entropy is added directly to secondary pool
  - Attacker can directly affect the generators output
  - Solution: always add entropy to primary pool

- The LRNG state reveals the previous LRNG output
  - Solution: switch order of operations (state update after LRNG output)
Security engineering

- No limits for reading from devices \(\Rightarrow\) denial of service attacks
  - Local attacker: simply reads from `/dev/random` device
  - Remote attacker: can establish many TCP connections
    (TCP/SYN requires 128 bits of random data from urandom pool)
  - Solution: definition of quotas per user/group

- Guessable passwords (applicable on disk-less systems)
  - First user-operation in a computer system is user login
  - LRNG state might be a deterministic function of initial user password
  - Solution: keyboard entropy based on timing (not on typed values)

- An adversary can create noise that directly affects the LRNG output
  - Full primary pool \(\Rightarrow\) entropy is added directly to secondary pool
  - Attacker can directly affect the generators output
  - Solution: always add entropy to primary pool

- The LRNG state reveals the previous LRNG output
  - Solution: switch order of operations (state update after LRNG output)
Security engineering

- No limits for reading from devices $\Rightarrow$ denial of service attacks
  - Local attacker: simply reads from /dev/random device
  - Remote attacker: can establish many TCP connections (TCP/SYN requires 128 bits of random data from urandom pool)
  - Solution: definition of quotas per user/group

- Guessable passwords (applicable on disk-less systems)
  - First user-operation in a computer system is user login
  - LRNG state might be a deterministic function of initial user password
  - Solution: keyboard entropy based on timing (not on typed values)

- An adversary can create noise that directly affects the LRNG output
  - Full primary pool $\Rightarrow$ entropy is added directly to secondary pool
  - Attacker can directly affect the generators output
  - Solution: always add entropy to primary pool

- The LRNG state reveals the previous LRNG output
  - Solution: switch order of operations (state update after LRNG output)
Real-world implications

- Almost all Linux distributions use the same kernel source
  - LRNG structure is thus very often the same
  - Small changes occur only within the system up and down times

- Initialization of LRNG
  - Constant parameters, time-of-day, disk operations and system events
  - Might be easily predicted (especially in systems without HDD)
  - Solution: LRNG simulates continuity along shutdowns and startups
    - Saving random seed by special script (no part of kernel)
    - Not applicable to all distributions (e.g., Knoppix, OpenWRT)

- OpenWRT – a Linux distribution for wireless routers
  - Very limited entropy sources (no keyboard, mouse, HDD)
  - Flash memory does not provide any entropy
  - The only entropy source are network interrupts
    - Easily observable (especially in wireless environment)
Real-world implications

- Almost all Linux distributions use the same kernel source
  - LRNG structure is thus very often the same
  - Small changes occur only within the system up and down times

- Initialization of LRNG
  - Constant parameters, time-of-day, disk operations and system events
  - Might be easily predicted (especially in systems without HDD)
  - Solution: LRNG simulates continuity along shutdowns and startups
    - Saving random seed by special script (no part of kernel)
    - Not applicable to all distributions (e.g., Knoppix, OpenWRT)

- OpenWRT – a Linux distribution for wireless routers
  - Very limited entropy sources (no keyboard, mouse, HDD)
  - Flash memory does not provide any entropy
  - The only entropy source are network interrupts
    - Easily observable (especially in wireless environment)
Real-world implications

- Almost all Linux distributions use the same kernel source
  - LRNG structure is thus very often the same
  - Small changes occur only within the system up and down times

- Initialization of LRNG
  - Constant parameters, time-of-day, disk operations and system events
  - Might be easily predicted (especially in systems without HDD)
  - Solution: LRNG simulates continuity along shutdowns and startups
    - Saving random seed by special script (no part of kernel)
    - Not applicable to all distributions (e.g., Knoppix, OpenWRT)

- OpenWRT – a Linux distribution for wireless routers
  - Very limited entropy sources (no keyboard, mouse, HDD)
  - Flash memory does not provide any entropy
  - The only entropy source are network interrupts
    - Easily observable (especially in wireless environment)
Final recommendation