Randomness generation

Daniel J. Bernstein, Tanja Lange

May 16, 2014
RDRAND: Just use it!

David Johnston, 2012 (emphasis added):

“That’s exactly why we put the new random number generator in our processors. To solve the chronic problem of security software systems lacking entropy. To provide secure random numbers even in VMs on blades. The rules of RNGs change when you have a 3Gbps source of entropy, which we do. You can over-engineer the downstream processing to ensure a reliable and sufficient supply under worst case assumptions. It’s not to solve ‘seeding’ issues. It provides both the entropy, the seeds and the PRNG in hardware. So you can replace the whole shebang and eliminate software PRNGs. Just use the output of the RDRAND instruction wherever you need a random number.”
The developers are listening

GitHub

Search

rdrand

Repositories 16

Code 46,932

Issues 70

We've found 46,932 code results

kmowery/rdrand – .gitignore
Last indexed 8 months ago
But does RDRAND actually work? RTFM


“Under heavy load, with multiple cores executing RDRAND in parallel, it is possible, though unlikely, for the demand of random numbers by software processes/threads to exceed the rate at which the random number generator hardware can supply them. This will lead to the RDRAND instruction returning no data transitorily. The RDRAND instruction indicates the occurrence of this rare situation by clearing the CF flag. . . . It is recommended that software using the RDRAND instruction to get random numbers retry for a limited number of iterations while RDRAND returns CF=0 and complete when valid data is returned, indicated with CF=1. This will deal with transitory underflows.”
#define SUCCESS 1
#define RETRY_LIMIT_EXCEEDED 0
#define RETRY_LIMIT 10

int get_random_64( unsigned __int 64 * arand) {
    int i;
    for (i = 0; i < RETRY_LIMIT; i++) {
        if(_rdrand64_step(arand) ) return SUCCESS;
    }
    return RETRY_LIMIT_EXCEEDED;
}
“Runtime failures in the random number generator circuitry or statistically anomalous data occurring by chance will be detected by the self test hardware and flag the resulting data as being bad. In such extremely rare cases, the RDRAND instruction will return no data instead of bad data.”

Intel’s DRNG Software Implementation Guide, Revision 1.1: “rare event that the DRNG fails during runtime”.

No quantification of “rare”.

Enter stay-dead state for one power-up out of every 10000?
Enter stay-dead state at certain voltages?
2013 Bernstein–Chang–Cheng–Chou–Heninger–Lange–van Someren “Factoring RSA keys from certified smart cards: Coppersmith in the wild” exploited such rare failures.
RDRAND conclusion: unsafe at any speed

If software keeps retrying: “busy loop”; software hangs.
RDRAND conclusion: unsafe at any speed

If software keeps retrying: “busy loop”; software hangs.
If software ignores EXCEDED: software uses “bad data”.

Cryptography Research: using RDRAND directly in applications is easy but “the most prudent approach is always to combine any other available entropy sources to avoid having a single point of failure.” This is exactly what BSD’s /dev/urandom does.
RDRAND conclusion: unsafe at any speed

If software keeps retrying: “busy loop”; software hangs.
If software ignores EXCEED: software uses “bad data”.
If software catches EXCEED: crypto dies.
RDRAND conclusion: unsafe at any speed

If software keeps retrying: “busy loop”; software hangs.
If software ignores EXCEEDEx: software uses “bad data”.
If software catches EXCEEDEx: crypto dies.
What’s the backup plan?
RDRAND conclusion: unsafe at any speed

If software keeps retrying: “busy loop”; software hangs.
If software ignores EXCEEDED: software uses “bad data”.
If software catches EXCEEDED: crypto dies.
What’s the backup plan? There is no backup plan!
RDRAND conclusion: unsafe at any speed

If software keeps retrying: “busy loop”; software hangs.
If software ignores EXCEEDED: software uses “bad data”.
If software catches EXCEEDED: crypto dies.
What’s the backup plan? There is no backup plan!

Cryptography Research:
using RDRAND directly in applications is easy but
“the most prudent approach is always to combine any other
available entropy sources to avoid having a single point of failure.”
RDRAND conclusion: unsafe at any speed

If software keeps retrying: “busy loop”; software hangs.
If software ignores EXCEEEDED: software uses “bad data”.
If software catches EXCEEEDED: crypto dies.

What’s the backup plan? There is no backup plan!

Cryptography Research: using RDRAND directly in applications is easy but “the most prudent approach is always to combine any other available entropy sources to avoid having a single point of failure.”

This is exactly what BSD’s /dev/urandom does.
Does RDRAND actually work properly?
Does RDRAND actually work properly?


(References from “Analysis of Intel’s Ivy Bridge Digital Random Number Generator Prepared for Intel” by Mike Hamburg, Paul Kocher, and Mark E. Marson. Cryptography Research, Inc.)
Design (from CRI report)

Figure 1: Block diagram of the Intel RNG (adapted from [7])

Raw Entropy Generation

Entropy Source (ES)

Shift Register (256 bits)

Digital Post-Processing

Health & Wellness Testing (OHT)

OSTE Queues (2x256 bits)

Deterministic Random Bit Generator (DRBG)

Conditioned Entropy (CE) (256 bits)

Output Generation & Reseeding

Output Buffers (4x128 bits)

Daniel J. Bernstein and Tanja Lange
Randomness generation
Entropy Source (from CRI report)

Figure 2: Entropy source for the Intel RNG (from [8])
“It uses the counter mode CTR_DRBG construction as defined in [2], with AES-128 as the block cipher.”
I’ve examined my own RNG with electron microscopes and picoprobes. So I and a number of test engineers know full well that the design hasn’t been subverted. For security critical systems, having multiple entropy sources is a good defense against a single source being subverted. But if an Intel processor were to be subverted, there are better things to attack, like the microcode or memory protection or caches. We put a lot of effort into keeping them secure, but as with any complex system it’s impossible to know that you’ve avoided all possible errors, so maintaining the security of platforms is an ongoing battle. [...] But the implication at the top of this thread is that we were leaned on by the government to undermine our own security features. I know for a fact that I was not leant on by anyone to do that. X9.82 took my contributions and NIST is taking about half my contributions, but maybe they’re slowly coming around to my way of thinking on online entropy testing. If I ultimately succeed in getting those specs to be sane, we better hope that I am sane.
Scary Paper of the Year: *Stealthy Dopant-Level Hardware Trojans*
by Becker, Regazzoni, Paar, and Burleson, CHES 2013

**Fig. 2.** Layout of the Trojan DFFR_X1 gate. The gate is only modified in the highlighted area by changing the dopant mask. The resulting Trojan gate has an output of $Q = V_{DD}$ and $QN = GND$. 
In case the hash function has some recognizable output pattern, we fold it in half. Thus, we always feed back twice as much data as we output.

```
hash.w[0] ^= hash.w[3];
hash.w[1] ^= hash.w[4];
hash.w[2] ^= rol32(hash.w[2], 16);
```

If we have a architectural hardware random number generator, mix that in, too.

```
for (i = 0; i < LONGS(EXTRACT_SIZE); i++) {
    unsigned long v;
    if (!arch_get_random_long(&v))
        break;
    hash.l[i] ^= v;
}
```

memcpy(out, &hash, EXTRACT_SIZE);
memset(&hash, 0, sizeof(hash));
“The way RDRAND is being used in kernels <= 3.12.3 allows it to cancel out the other entropy. See extract_buf().”
“if I make RDRAND return [EDX] ^ 0x41414141, /dev/urandom output will be all ’A’.” Full thread
/*
 * If we have an architectural hardware random number
 * generator, use it for SHA’s initial vector
 */
sha_init(hash.w);
for (i = 0; i < LONGS(20); i++) {
    unsigned long v;
    if (!arch_get_random_long(&v))
        break;
    hash.l[i] = v;
}
/* Generate a hash across the pool,
 * 16 words (512 bits) at a time */
spin_lock_irqsave(&r->lock, flags);
for (i = 0; i < r->poolinfo->poolwords; i += 16)
    sha_transform(hash.w, (__u8 *)(r->pool + i), workspace);
<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Author</th>
<th>Commit Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-12-17</td>
<td>21:16</td>
<td>Theodore Ts'o</td>
<td>o [dev] [origin/dev] random: use the architectural HWRNG for~</td>
</tr>
<tr>
<td>2013-12-06</td>
<td>21:28</td>
<td>Greg Price</td>
<td>o random: clarify bits/bytes in wakeup thresholds</td>
</tr>
<tr>
<td>2013-12-07</td>
<td>09:49</td>
<td>Greg Price</td>
<td>o random: entropy_bytes is actually bits</td>
</tr>
<tr>
<td>2013-12-05</td>
<td>19:32</td>
<td>Greg Price</td>
<td>o random: simplify accounting code</td>
</tr>
<tr>
<td>2013-12-05</td>
<td>19:19</td>
<td>Greg Price</td>
<td>o random: tighten bound on random_read_wakeup_thresh</td>
</tr>
<tr>
<td>2013-11-29</td>
<td>20:09</td>
<td>Greg Price</td>
<td>o random: forget lock in lockless accounting</td>
</tr>
<tr>
<td>2013-11-29</td>
<td>15:56</td>
<td>Greg Price</td>
<td>o random: simplify accounting logic</td>
</tr>
<tr>
<td>2013-11-29</td>
<td>15:50</td>
<td>Greg Price</td>
<td>o random: fix comment on &quot;account&quot;</td>
</tr>
<tr>
<td>2013-11-29</td>
<td>15:02</td>
<td>Greg Price</td>
<td>o random: simplify loop in random_read</td>
</tr>
<tr>
<td>2013-11-29</td>
<td>14:59</td>
<td>Greg Price</td>
<td>o random: fix description of get_random_bytes</td>
</tr>
<tr>
<td>2013-11-29</td>
<td>14:58</td>
<td>Greg Price</td>
<td>o random: fix comment on proc_do_uuid</td>
</tr>
<tr>
<td>2013-11-29</td>
<td>14:58</td>
<td>Greg Price</td>
<td>o random: fix typos / spelling errors in comments</td>
</tr>
<tr>
<td>2013-11-16</td>
<td>10:19</td>
<td>Linus Torvalds</td>
<td>M-</td>
</tr>
<tr>
<td>2013-11-03</td>
<td>18:24</td>
<td>Theodore Ts'o</td>
<td>o [random_for_linus] random: add debugging code to detect ~</td>
</tr>
<tr>
<td>2013-11-03</td>
<td>16:40</td>
<td>Theodore Ts'o</td>
<td>o random: initialize the last_time field in struct timer_r~</td>
</tr>
<tr>
<td>2013-11-03</td>
<td>07:56</td>
<td>Theodore Ts'o</td>
<td>o random: don’t zap entropy count in rand_initialize()</td>
</tr>
<tr>
<td>2013-11-03</td>
<td>06:54</td>
<td>Theodore Ts'o</td>
<td>o random: printk notifications for urandom pool initialize~</td>
</tr>
<tr>
<td>2013-11-03</td>
<td>00:15</td>
<td>Theodore Ts'o</td>
<td>o random: make add_timer_randomness() fill the nonblocking~</td>
</tr>
<tr>
<td>2013-10-03</td>
<td>12:02</td>
<td>Theodore Ts'o</td>
<td>o random: convert DEBUG_ENT to tracepoints</td>
</tr>
<tr>
<td>2013-10-03</td>
<td>01:08</td>
<td>Theodore Ts'o</td>
<td>o random: push entropy to the output pools</td>
</tr>
<tr>
<td>2013-10-02</td>
<td>21:10</td>
<td>Theodore Ts'o</td>
<td>o random: drop trickle mode</td>
</tr>
<tr>
<td>2013-09-22</td>
<td>16:04</td>
<td>Theodore Ts'o</td>
<td>o random: adjust the generator polynomials in the mixing f~</td>
</tr>
<tr>
<td>2013-09-22</td>
<td>15:24</td>
<td>Theodore Ts'o</td>
<td>o random: speed up the fast_mix function by a factor of fo~</td>
</tr>
<tr>
<td>2013-09-22</td>
<td>15:14</td>
<td>Theodore Ts'o</td>
<td>o random: cap the rate which the /dev/urandom pool gets re~</td>
</tr>
<tr>
<td>2013-09-21</td>
<td>19:42</td>
<td>Theodore Ts'o</td>
<td>o random: optimize the entropy_store structure</td>
</tr>
<tr>
<td>2013-09-12</td>
<td>14:27</td>
<td>Theodore Ts'o</td>
<td>o random: optimize spinlock use in add_device_randomness()</td>
</tr>
<tr>
<td>2013-09-12</td>
<td>14:10</td>
<td>Theodore Ts'o</td>
<td>o random: fix the tracepoint for get_random_bytes(_arch)</td>
</tr>
<tr>
<td>2013-09-10</td>
<td>23:16</td>
<td>H. Peter Anvin</td>
<td>o random: account for entropy loss due to overwrites</td>
</tr>
<tr>
<td>2013-09-10</td>
<td>23:16</td>
<td>H. Peter Anvin</td>
<td>o random: allow fractional bits to be tracked</td>
</tr>
<tr>
<td>2013-09-21</td>
<td>18:06</td>
<td>Theodore Ts'o</td>
<td>o random: statically compute poolbitshift, poolbytes, pool~</td>
</tr>
<tr>
<td>2013-09-21</td>
<td>23:16</td>
<td>H. Peter Anvin</td>
<td>o random: mix in architectural randomness earlier in extra~</td>
</tr>
<tr>
<td>2013-11-11</td>
<td>12:20</td>
<td>Hannes Frederic S~</td>
<td>random32: add prandom_reseed_late() and call when nonblo~</td>
</tr>
<tr>
<td>2013-10-10</td>
<td>12:31</td>
<td>Linus Torvalds</td>
<td>M-</td>
</tr>
<tr>
<td>2013-09-21</td>
<td>13:58</td>
<td>Theodore Ts'o</td>
<td>o random: allow architectures to optionally define random~</td>
</tr>
<tr>
<td>2013-09-10</td>
<td>10:52</td>
<td>Theodore Ts'o</td>
<td>o random: run random_int_secret_init() run after all late~</td>
</tr>
</tbody>
</table>
| 2013-08-30  | 09:39  | Martin Schwidefsky | o | Remove GENERIC_HARDIRQ config option
What would we like to see?

- Cryptographers can help here!
- Easy part: Stream cipher generates randomness from seed. With big seed, safe to have output overwrite old seed.
- Hard part: Need comprehensible mechanism to securely merge entropy sources into seed.
- Some sources are bad. Is full hashing really necessary?
- Some sources are influenced or controlled by attacker. Is protection against malice possible?
- Maybe helpful: Some malicious sources have limited time and space. Concatenate independent hashes of several sources, apply many rounds of wide permutation, then truncate?