random.c

Inside the Linux Kernel RNG

Presented by Jason A. Donenfeld

September 13, 2022

Linux Plumbers Conference
Dublin, Ireland

Who Am I?

- Jason Donenfeld, also known as **zx2c4**.
- Background in exploitation, kernel vulnerabilities, crypto vulnerabilities, and been doing kernel-related development for a long time.
- Have been working on WireGuard for the last few years, where high quality cryptographic randomness is important.
- Put a bit of time into Linux's random number generator this year.

Old Code

- The kernel's RNG lives in random.c, which dates back to version 1.3.30.
- Copyright date is 1994.
- Considering when it was written, actually not so bad.
- Has held up relatively well, all things considered.
- But…

```
198  static struct timer_rand_state irq_timer_state[NR_IRQS];
199
200  #ifndef MIN
201  #define MIN(a,b) (((a) < (b)) ? (a) : (b))
202  #endif
203
204  static void flush_random(struct random_bucket *random_state)
205  {
206          random_state->add_ptr = 0;
207          random_state->bit_length = random_state->length * 8;
208          random_state->entropy_count = 0;
209          random_state->delay_mix = 0;
210  }
211
212  void rand_initialize(void)
213  {
214          random_state.length = RANDPOOL;
215          random_state.pool = random_pool;
216          flush_random(&random_state);
217  }
218
219  /*
220   * MD5 transform algorithm, taken from code written by Colin Plumb,
221   * and put into the public domain
222   */
223
224  /* The four core functions - F1 is optimized somewhat */
225
226  /* #define F1(x, y, z) (x & y | ~x & z) */
227  #define F1(x, y, z) (z ^ (x & (y ^ z)))
228  #define F2(x, y, z) F1(z, x, y)
229  #define F3(x, y, z) (x ^ y ^ z)
230  #define F4(x, y, z) (y ^ (x | ~z))
231
232  /* This is the central step in the MD5 algorithm. */
233  #define MD5STEP(f, w, x, y, z, data, s) \
234          ( w += f(x, y, z) + data,  w = w<<s | w>>(32-s),  w += x )
235
236  /*
237   * The core of the MD5 algorithm, this alters an existing MD5 hash to
238   * reflect the addition of 16 longwords of new data.  MD5Update blocks
239   * the data and converts bytes into longwords for this routine.
240   */
241  static void MD5Transform(__u32 buf[4],
242                           __u32 const in[16])
243  {
244          __u32 a, b, c, d;
245
246          a = buf[0];
247          b = buf[1];
248          c = buf[2];
249          d = buf[3];
250
251          MD5STEP(F1, a, b, c, d, in[ 0]+0xd76aa478,  7);
252          MD5STEP(F1, d, a, b, c, in[ 1]+0xe8c7b756, 12);
253          MD5STEP(F1, c, d, a, b, in[ 2]+0x242070db, 17);
254          MD5STEP(F1, b, c, d, a, in[ 3]+0xc1bdceee, 22);
```

Old Code

- Over time, lots of complexity was introduced.
- Differing code styles, old code styles, funny types.
- Many conflicting ideas about its design, many piecemeal approaches.
- Very *practical* code, needing to serve an ever growing number of users with varying requirements.
- But still old and jumbled.

```
/*
 * random.c -- A strong random number generator
 *
 * Version 0.92, last modified 21-Sep-95
 *
 * Copyright Theodore Ts'o, 1994, 1995.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, and the entire permission notice in its entirety,
 *    including the disclaimer of warranties.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. The name of the author may not be used to endorse or promote
 *    products derived from this software without specific prior
 *    written permission.
 *
 * ALTERNATIVELY, this product may be distributed under the terms of
 * the GNU Public License, in which case the provisions of the GPL are
 * required INSTEAD OF the above restrictions.  (This clause is
 * necessary due to a potential bad interaction between the GPL and
 * the restrictions contained in a BSD-style copyright.)
 *
 * THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESS OR IMPLIED
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 * OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT,
 * INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
 * (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
 * SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
 * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
 * OF THE POSSIBILITY OF SUCH DAMAGE.
 */

/*
 * (now, with legal B.S. out of the way.....)
 *
 * This routine gathers environmental noise from device drivers, etc.,
```

Old Code Example: LFSR Proliferation

- At some point, performance concerns (or security concerns?) lead to replacing the primary mixing function with a faster LFSR.
  - Used to just be MD5. We know *now* that this is a bad hash function, but at least then the motivation (cryptographically secure hash function) was okay.
  - Also, this LFSR isn't especially fast.
- This LFSR was tweaked over the years.
- Later, other LFSRs were added for various purposes.

```
536 static void _mix_pool_bytes(struct entropy_store *r, const void *in,
537                             int nbytes)
538 {
539         unsigned long i, tap1, tap2, tap3, tap4, tap5;
540         int input_rotate;
541         int wordmask = r->poolinfo->poolwords - 1;
542         const char *bytes = in;
543         __u32 w;
544
545         tap1 = r->poolinfo->tap1;
546         tap2 = r->poolinfo->tap2;
547         tap3 = r->poolinfo->tap3;
548         tap4 = r->poolinfo->tap4;
549         tap5 = r->poolinfo->tap5;
550
551         input_rotate = r->input_rotate;
552         i = r->add_ptr;
553
554         /* mix one byte at a time to simplify size handling and churn faster */
555         while (nbytes--) {
556                 w = rol32(*bytes++, input_rotate);
557                 i = (i - 1) & wordmask;
558
559                 /* XOR in the various taps */
560                 w ^= r->pool[i];
561                 w ^= r->pool[(i + tap1) & wordmask];
562                 w ^= r->pool[(i + tap2) & wordmask];
563                 w ^= r->pool[(i + tap3) & wordmask];
564                 w ^= r->pool[(i + tap4) & wordmask];
565                 w ^= r->pool[(i + tap5) & wordmask];
566
567                 /* Mix the result back in with a twist */
568                 r->pool[i] = (w >> 3) ^ twist_table[w & 7];
569
570                 /*
571                  * Normally, we add 7 bits of rotation to the pool.
572                  * At the beginning of the pool, add an extra 7 bits
573                  * rotation, so that successive passes spread the
574                  * input bits across the pool evenly.
575                  */
576                 input_rotate = (input_rotate + (i ? 7 : 14)) & 31;
577         }
578
579         r->input_rotate = input_rotate;
580         r->add_ptr = i;
581 }
```

```
522  static __u32 const twist_table[8] = {
523          0x00000000, 0x3b6e20c8, 0x76dc4190, 0x4db26158,
524          0xedb88320, 0xd6d6a3e8, 0x9b64c2b0, 0xa00ae278 };
```

```
426  static const struct poolinfo {
427          int poolbitshift, poolwords, poolbytes, poolfracbits;
428  #define S(x) ilog2(x)+5, (x), (x)*4, (x) << (ENTROPY_SHIFT+5)
429          int tap1, tap2, tap3, tap4, tap5;
430  } poolinfo_table[] = {
431          /* was: x^128 + x^103 + x^76 + x^51 +x^25 + x + 1 */
432          /* x^128 + x^104 + x^76 + x^51 +x^25 + x + 1 */
433          { S(128),       104,    76,     51,     25,     1 },
434  };
```

- What is this code doing?
- Can we break it without having to answer that question?

Old Code Example: LFSR Mischief with SMT

- Rather than having to think, let the computer think for us.
- Code it up in PyZ3, output SMT2, feed it to Boolector.
- Give the pool some inputs and control its state.
- What does this attack even correspond to?
  - (Does anybody actually care about this security model? More later.)

```
POOL_WORDS = 128
POOL_WORDMASK = POOL_WORDS - 1
POOL_BYTES = POOL_WORDS * 4
POOL_BITS = POOL_BYTES * 8

# x^128 + x^104 + x^76 + x^51 +x^25 + x + 1
POOL_TAP1 = 104
POOL_TAP2 = 76
POOL_TAP3 = 51
POOL_TAP4 = 25
POOL_TAP5 = 1

def mix_pool_byte(input_byte, input_pool_data, input_pool_index, input_pool_rotate):
    global twist_table
    w = RotateLeft(ZeroExt(24, input_byte), input_pool_rotate)
    i = (input_pool_index - 1) & POOL_WORDMASK
    w ^= input_pool_data[i]
    w ^= input_pool_data[(i + POOL_TAP1) & POOL_WORDMASK]
    w ^= input_pool_data[(i + POOL_TAP2) & POOL_WORDMASK]
    w ^= input_pool_data[(i + POOL_TAP3) & POOL_WORDMASK]
    w ^= input_pool_data[(i + POOL_TAP4) & POOL_WORDMASK]
    w ^= input_pool_data[(i + POOL_TAP5) & POOL_WORDMASK]
    w = LShR(w, 1) ^ (0xedb88320 & -(w & 1))
    w = LShR(w, 1) ^ (0xedb88320 & -(w & 1))
    w = LShR(w, 1) ^ (0xedb88320 & -(w & 1))
    input_pool_data[i] = w
    return input_pool_data, i, (input_pool_rotate + (14 if i == 0 else 7)) & 31

def popcount(word):
    return Sum([ZeroExt(5, Extract(i, i, word)) for i in range(32)])

def popcount_all(inputs):
    return Sum([ZeroExt(7, popcount(word)) for word in inputs])

pool, inputs, idx, rot = [BitVec("pool_%02d" % i, 32) for i in range(128)], [], 0, 0
initial_pool = pool[:]
```

Old Code Example: LFSR Mischief with Linear Algebra

- Rather than letting the computer think for us, might as well make some human effort.
- LFSRs are, by definition, linear functions.
  - Take a state vector, S, multiply it by some matrix, A, add input vector, X, to produce new state. $S \leftarrow AS \oplus X$
- So actually we can have Magma compute the whole thing a lot faster for us.
- Which reminds me…

Aside: OpenBSD

- Long ago, Linux's LFSR was selectable.
- There were many choices, big and small.
- The polynomials choices were selected to be primitive, which is what you want to have a max-period LFSR.
- Later a "twist" was added in the form of the CRC-32 polynomial, forming an extension field.

```
/*
 * A pool of size .poolwords is stirred with a primitive polynomial
 * of degree .poolwords over GF(2).  The taps for various sizes are
 * defined below.  They are chosen to be evenly spaced (minimum RMS
 * distance from evenly spaced; the numbers in the comments are a
 * scaled squared error sum) except for the last tap, which is 1 to
 * get the twisting happening as fast as possible.
 */
static struct poolinfo {
        int     poolwords;
        int     tap1, tap2, tap3, tap4, tap5;
} poolinfo_table[] = {
        /* x^2048 + x^1638 + x^1231 + x^819 + x^411 + x + 1  -- 115 */
        { 2048, 1638,   1231,   819,    411,    1 },

        /* x^1024 + x^817 + x^615 + x^412 + x^204 + x + 1 -- 290 */
        { 1024, 817,    615,    412,    204,    1 },
#if 0                           /* Alternate polynomial */
        /* x^1024 + x^819 + x^616 + x^410 + x^207 + x^2 + 1 -- 115 */
        { 1024, 819,    616,    410,    207,    2 },
#endif

        /* x^512 + x^411 + x^308 + x^208 + x^104 + x + 1 -- 225 */
        { 512,  411,    308,    208,    104,    1 },
#if 0                           /* Alternates */
        /* x^512 + x^409 + x^307 + x^206 + x^102 + x^2 + 1 -- 95 */
        { 512,  409,    307,    206,    102,    2 },
        /* x^512 + x^409 + x^309 + x^205 + x^103 + x^2 + 1 -- 95 */
        { 512,  409,    309,    205,    103,    2 },
#endif

        /* x^256 + x^205 + x^155 + x^101 + x^52 + x + 1 -- 125 */
        { 256,  205,    155,    101,    52,     1 },

        /* x^128 + x^103 + x^76 + x^51 +x^25 + x + 1 -- 105 */
        { 128,  103,    76,     51,     25,     1 },
#if 0   /* Alternate polynomial */
        /* x^128 + x^103 + x^78 + x^51 + x^27 + x^2 + 1 -- 70 */
        { 128,  103,    78,     51,     27,     2 },
#endif

        /* x^64 + x^52 + x^39 + x^26 + x^14 + x + 1 -- 15 */
        { 64,   52,     39,     26,     14,     1 },

        /* x^32 + x^26 + x^20 + x^14 + x^7 + x + 1 -- 15 */
        { 32,   26,     20,     14,     7,      1 },

        { 0,    0,      0,      0,      0,      0 },
};
```

```
 static __u32 const twist_table[8] = {
         0x00000000, 0x3b6e20c8, 0x76dc4190, 0x4db26158,
         0xedb88320, 0xd6d6a3e8, 0x9b64c2b0, 0xa00ae278 };
```

```
w = rol32(*bytes++, input_rotate);
i = (i - 1) & wordmask;

/* XOR in the various taps */
w ^= r->pool[i];
w ^= r->pool[(i + tap1) & wordmask];
w ^= r->pool[(i + tap2) & wordmask];
w ^= r->pool[(i + tap3) & wordmask];
w ^= r->pool[(i + tap4) & wordmask];
w ^= r->pool[(i + tap5) & wordmask];

/* Mix the result back in with a twist */
r->pool[i] = (w >> 3) ^ twist_table[w & 7];

/*
 * Normally, we add 7 bits of rotation to the pool.
 * At the beginning of the pool, add an extra 7 bits
 * rotation, so that successive passes spread the
 * input bits across the pool evenly.
 */
input_rotate = (input_rotate + (i ? 7 : 14)) & 31;
```

Linux Source Code

Aside: OpenBSD

- OpenBSD grabbed one of the primitive polynomials from that list – the biggest one.
- And it used the same CRC-32 “twist table” polynomial as an extension field.
- **Problem:** though the polynomial it chose is primitive, it is not so when used in the CRC-32 extension field!
- **…so we can just factor it…**

```
/*
 * Stirring polynomials over GF(2) for various pool sizes. Used in
 * add_entropy_words() below.
 *
 * The polynomial terms are chosen to be evenly spaced (minimum RMS
 * distance from evenly spaced; except for the last tap, which is 1 to
 * get the twisting happening as fast as possible.
 *
 * The resultant polynomial is:
 *   2^POOLWORDS + 2^POOL_TAP1 + 2^POOL_TAP2 + 2^POOL_TAP3 + 2^POOL_TAP4 + 1
 */
#define POOLWORDS       2048
#define POOLBYTES       (POOLWORDS*4)
#define POOLMASK        (POOLWORDS - 1)
#define POOL_TAP1       1638
#define POOL_TAP2       1231
#define POOL_TAP3       819
#define POOL_TAP4       411
```

```
/* derived from IEEE 802.3 CRC-32 */
static const u_int32_t twist_table[8] = {
        0x00000000, 0x3b6e20c8, 0x76dc4190, 0x4db26158,
        0xedb88320, 0xd6d6a3e8, 0x9b64c2b0, 0xa00ae278
};
static u_int    entropy_add_ptr;
static u_char   entropy_input_rotate;

for (; n--; buf++) {
        u_int32_t w = (*buf << entropy_input_rotate) |
            (*buf >> ((32 - entropy_input_rotate) & 31));
        u_int i = entropy_add_ptr =
            (entropy_add_ptr - 1) & POOLMASK;
        /*
         * Normally, we add 7 bits of rotation to the pool.
         * At the beginning of the pool, add an extra 7 bits
         * rotation, so that successive passes spread the
         * input bits across the pool evenly.
         */
        entropy_input_rotate =
            (entropy_input_rotate + (i ? 7 : 14)) & 31;

        /* XOR pool contents corresponding to polynomial terms */
        w ^= entropy_pool[(i + POOL_TAP1) & POOLMASK] ^
             entropy_pool[(i + POOL_TAP2) & POOLMASK] ^
             entropy_pool[(i + POOL_TAP3) & POOLMASK] ^
             entropy_pool[(i + POOL_TAP4) & POOLMASK] ^
             entropy_pool[(i + 1) & POOLMASK] ^
             entropy_pool[i]; /* + 2^POOLWORDS */

        entropy_pool[i] = (w >> 3) ^ twist_table[w & 7];
}
```

OpenBSD Source Code

Aside: OpenBSD

- **…so we can just factor it…**
- Pretty small terms means very small cycles?

```
zx2c4@thinkpad ~ $ ssh -p 19230 localhost -t Magma/magma
Magma              Mon Sep 12 2022 02:16:43 on localhost
Type ? for help.  Type <Ctrl>-D to quit.
> Q<x> := ExtensionField<GF(2), X|X^32 + X^26 + X^23 + X^22 + X^16 + X^12 + X^11 + X^10 + X^8 + X^7 + X^5 + X^4 + X^2 + X + 1>;
> R<X> := PolynomialRing(Q);
> Factorization(( x^3*(X^2048 + X^1638 + X^1231 + X^819 + X^411 + X + 1 - 1) + 1 ));
[
    <X + x^30 + x^26 + x^25 + x^23 + x^22 + x^19 + x^18 + x^17 + x^16 + x^15 + x^11 + x^10 + x^8 + x^6 + x^3 + x, 1>,
    <X^31 + (x^30 + x^29 + x^28 + x^27 + x^23 + x^21 + x^20 + x^19 + x^18 + x^13 + x^12 + x^10 + x^9 + x^8 + x^7 + x^5 + x^3 + x +
        x^18 + x^16 + x^15 + x^14 + x^13 + x^12 + x^11 + x^9 + x^8 + x^2 + x + 1)*X^29 + (x^30 + x^28 + x^27 + x^26 + x^23 + x^20
        x)*X^28 + (x^30 + x^29 + x^28 + x^27 + x^22 + x^21 + x^20 + x^18 + x^17 + x^15 + x^14 + x^13 + x^12 + x^10 + x^9 + x^8 + x
        + x^16 + x^13 + x^11 + x^9 + x^6 + x^4 + x^2)*X^26 + (x^30 + x^29 + x^27 + x^26 + x^24 + x^23 + x^20 + x^19 + x^18 + x^10
        x^24 + x^22 + x^19 + x^17 + x^15 + x^14 + x^8 + x^6 + x^3 + x^2 + 1)*X^24 + (x^29 + x^28 + x^27 + x^25 + x^22 + x^17 + x^1
        (x^31 + x^30 + x^27 + x^26 + x^24 + x^20 + x^17 + x^16 + x^13 + x^7 + x^6 + x^4 + x)*X^22 + (x^26 + x^21 + x^16 + x^15 + x
        + x^27 + x^26 + x^25 + x^18 + x^13 + x^11 + x^10 + x^8 + x^2 + x + 1)*X^20 + (x^31 + x^28 + x^27 + x^26 + x^21 + x^19 + x^
```

Bigger Picture

- LFSR mischief is just one fun attribute of the old code.
  - But doesn't really matter. It's not there anymore.
  - Depending on threat model (more later), maybe not even a problem in Linux's usage.
  - But… why not just use a cryptographic hash function?
- So, let's dive into how the RNG works *now* instead.
- First, how did we get there?

Incrementalism

- There have been attempts at wholesale replacement of the RNG.
- Massive sweeping replacement isn’t generally how Linux development works.
  - Difficult to review.
  - Feather ruffling.
  - Tends to forget about various practical subtle aspects that went into original code.
- Every individual commit motivated by *specific reason*, and backed up with **code archeology**.
  - Understand the history and intent of existing code before changing it.
  - Don’t inadvertently break things.

Incrementalism

```
zx2c4@thinkpad ~/Projects/random-linux $ git log --oneline --since=2021-12-01 drivers/char/random.c | wc -l
147
```

```
zx2c4@thinkpad ~/Projects/random-linux $ git log --pretty=format:%B --since=2021-12-01 drivers/char/random.c | grep -v '^[A-Z][A-Za-z-]\+:' | grep -v '^$' | wc
   1975   18349  118274
```

```
zx2c4@thinkpad ~/Projects/random-linux $ cloc drivers/char/random.c
       1 text file.
       1 unique file.
       0 files ignored.

github.com/AlDanial/cloc v 1.90  T=0.01 s (188.8 files/s, 299849.4 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C                                1            175            465            948
-------------------------------------------------------------------------------
```

Diving In – Several Components

- Entropy collection
- Entropy extraction
- Entropy expansion
- Entropy sources
- “Are we initialized yet?”
- Userspace interfaces

Diving In – Several Components

- **Entropy collection**
- Entropy extraction
- Entropy expansion
- Entropy sources
- “Are we initialized yet?”
- Userspace interfaces

Entropy Collection

- Replaces the prior LFSR.
- We simply use BLAKE2s – a cryptographic hash function.
  - Can operate in “HASH” mode or “PRF” mode, which takes a key.
  - “PRF” mode essentially just hashes a block with the key first.
- ```
  blake2s_update(&entropy_pool, input1, sizeof(input1));
  blake2s_update(&entropy_pool, input2, sizeof(input2));
  blake2s_update(&entropy_pool, input3, sizeof(input3));
  blake2s_update(&entropy_pool, input4, sizeof(input4));
  …
  ```
- Sort of basic and dumb.
- There’s actually academic literature trying to show that computational hash functions like this are good at this task.
- Doesn’t have the pitfalls that the LFSR had.
  ```

Diving In – Several Components

- Entropy collection
- **Entropy extraction**
- Entropy expansion
- Entropy sources
- “Are we initialized yet?”
- Userspace interfaces

Entropy Extraction – Old

- Old code called `SHA1(LFSR-Pool)`,
- And then xor'd one half with the other – same as just truncating.
- SHA1!
- In theory, fine here. But also, we can do much, much better.

Entropy Extraction – New

- Generate a block of RDSEED/RDRAND as a “salt”.
  - Really just an additional input that we can generate outside of locks.
  - More on RDSEED/RDRAND usage later.
- ```
  temp = blake2s_final(&entropy_pool);
  output1 = blake2s(key=temp, RDSEED-Block || 0x01);
  output2 = blake2s(key=temp, RDSEED-Block || 0x02);
  ```
- ```
  // Carry forward entropy into next collection.
  blake2s_init(&entropy_pool, key=output1);
  ```
- ```
  // The “extracted” data.
  return output2;
  ```
  ```

Diving In – Several Components

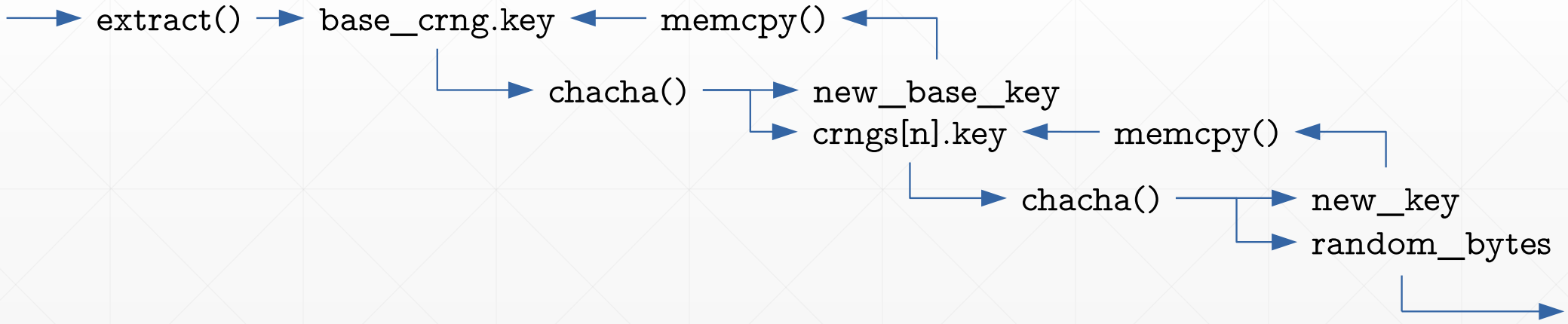
- Entropy collection
- Entropy extraction
- **Entropy expansion**
- Entropy sources
- “Are we initialized yet?”
- Userspace interfaces

Entropy Expansion

- Give the extracted data to ChaCha20 and let it rip.
- “Fast key-erasure RNG”
- One goal is forward secrecy: if you compromise the state of the RNG *now*, you can’t figure out values it generated in the past.
- chacha20(key) → “unlimited” stream of random bytes derived from “key”.
- “key” is 32 bytes. So use the first 32 bytes of that “unlimited” stream to immediately overwrite “key”.

Entropy Expansion

- Another goal is speed: this is the primary code used by `getrandom()`, `/dev/[u]random`, `get_random_bytes()`, etc.
- Per-CPU ChaCha20 instance for lockless expansion.
- In the previous “extraction” phase, we are yielding one seed. But this seed needs to potentially be extracted, in parallel, on N CPUs.
- First key goes to “base” instance, which gets expanded for each per-CPU instance.



Diving In – Several Components

- Entropy collection
- Entropy extraction
- Entropy expansion
- **Entropy sources**
- “Are we initialized yet?”
- Userspace interfaces

Entropy Sources – `add_device_randomness()`

- `add_device_randomness()` is used for any inputs that *might* have some "randomness" some of the time.
- It's used from various odd drivers here and there. For example:
  - Ingests DMI tables at bootup.
  - MMC card serial numbers.
  - Udev USB serial numbers.
  - MAC Addresses.
  - Wall clock & wall clock changes.
- It can't hurt; it can only help.
- If you're a driver writer and think you have a nice idea on some data you could use, go for it! (Just CC me on the patch.)

Entropy Sources – `add_hwgenerator_randomness()`

- `add_hwgenerator_randomness()` is used for dedicated hardware RNGs, with drivers in the hwrng framework.
- The hwrng framework is a bit messy, but from random.c's perspective it doesn't matter.
- Hands random.c some random bytes, and some numerical indication of how random it is (more on that that later).
- Private interface; don't use this from your own code.

Entropy Sources – `add_bootloader_randomness()`

- `add_bootloader_randomness()` is used by device tree, EFI, and other drivers.
- Receive random seeds from bootloader or early firmware.
- Preferably runs extremely early in Linux's bootup.
- Currently called from:
  - drivers/of/fdt.c – device tree
  - drivers/firmware/efi/efi.c – EFI (which will also work with `systemd-boot`)
  - arch/x86/kernel/setup.c – x86 setup_data linked list
  - arch/m68k/virt/config.c – m68k boot protocol
  - arch/um/kernel/um_arch.c – UML invokes `getrandom()` syscall
- Does your favorite architecture's firmware or bootloader provide anything useful not covered?

Entropy Sources – `add_vmfork_randomness()`

- `add_vmfork_randomness()` is triggered when the hypervisor forks.
- The hypervisor provides some unique ID that we use to reseed the RNG when those forks happen.
- (Various other events cause reseeding too, such as on wakeup from hibernation.)
- Covered in a talk yesterday.

Entropy Sources – `add_{input,disk}_randomness()`

- `add_input_randomness()` is triggered by key presses and mouse movements.
  - Handles auto-repeat in a somewhat naive way.
- `add_disk_randomness()` is triggered by rotational disk operations.
- Both of these try to estimate entropy, *the oldest surviving code in random.c that I haven't rewritten.* (More on entropy estimation later.)
- Takes the obvious set of inputs (e.g. key press code), but the primary input for these is actually a cycle counter via `random_get_entropy()`.

Entropy Sources – `add_interrupt_randomness()`

- `add_interrupt_randomness()` is called by the interrupt handler.
- It is often times the primary and most important source!
- Takes the obvious set of inputs (e.g. key press code), but the primary input for these is actually a cycle counter via `random_get_entropy()`.
- Since it runs in hard IRQ context, it needs to be very fast.
- Accumulates data from several interrupts using the `fast_mix()` function.
- After accumulating data from a certain number of interrupts, it queues up a worker to dump these into the main entropy pool like the other sources do.

`fast_mix()`

- Goal is to compress 2 longs at a time into a state consisting of 4 longs.
- Currently based on SipHash's permutation in a sponge construction.
  - Exploring other sponges now that might be faster / better.
- Used to be a *custom ARX permutation from anonymous contributor extraordinaire George Spelvin!*
  - Any time you have anybody contributing "hand rolled crypto", be wary, with an anonymous identity being even more eyebrow raising.
  - But I love George Spelvin! If you're still out there, get in touch with me!

`random_get_entropy()`

- Expands to `get_cycles()` on many platforms.
- Expands to something specific and custom on many other platforms.
- Falls back to timekeeping.c's raw monotonic counter source as a final resort.
- Since this is the **primary input material** in a great many circumstances, this needs to be as high resolution as possible, while still being fast.
- If it's not implemented on your favorite architecture/platform in a satisfactory way, let's fix it with some particular arch-specific code!

Entropy Sources – The Linus Jitter Dance

- The “Linus Jitter Dance” is used at early boot time if nothing else has initialized the RNG.
- In a loop,
  - Input a sample from `random_get_entropy()`,
  - If no timer is currently armed, arm a timer callback 1 jiffy from now,
  - After that timer callback executes ~256 times, break out of the loop.
    - The loop itself iterates way more than the timer callback executes.
    - Around 512 jiffies worth of loops.
- Relies on `random_get_entropy()` being high quality.
  - Adjusts for lower quality cycle counters to a degree, but there’s a floor on minimum quality.
  - Architecture people: let’s think about ways to improve `random_get_entropy()` on limited platforms.
- Unrigorous voodoo, but not obviously terrible either.

Entropy Sources – RDSEED / RDRAND

- Old code would incorporate this in a variety of questionable ways:
  - For example, in some instances, it would xor RDRAND into the output stream directly.
  - Depending on how paranoid you are, this is maybe not so good…
- New code *always* puts all RDRAND values through the hash function when entering the entropy pool somehow.
  - Means that RDRAND itself can't backdoor the pool without having a hash preimage.

Diving In – Several Components

- Entropy collection
- Entropy extraction
- Entropy expansion
- Entropy sources
- **“Are we initialized yet?”**
- Userspace interfaces

“Are we initialized yet?”

- It’s important that the RNG has 256 bits of entropy before it generates output.
- If the RNG has, say, only 8 bits of entropy, then it’s trivial to bruteforce its state and predict all output.
- Unless we have a physical guarantee from hardware manufacturers or seed files or elsewhere that a given data source has N bits of entropy, we can’t know for certain the entropy of any given input.
- **Heuristics** are the best we can do.
  - Unfortunate to admit, but this is essentially the case.

Entropy Counting & Estimation

- We assume ~64 interrupts have 1 bit of entropy.
- We assume disk timings and mouse/keyboard events have a variable amount of entropy.
  - There's an ancient entropy estimation algorithm that looks at third degree differentials to make some guess.
  - It's complete rubbish. Maybe it should be replaced with something smarter. But it's not altogether bad either. It's just a *heuristic.*
  - There are more scientific things we could do like computing FFTs on the fly and trying to get even distributions, but it's not clear that'd help in practice.
- We assume hardware RNGs have however much entropy the driver/hardware says it has.
- We assume bootloader seeds and RDRAND have full entropy (unless disabled via a command line parameter).
- We assume jitter has some amount of entropy relative to the scheduler in the Linus Jitter Dance.

Entropy Counting & Estimation

- Only ever relevant at boot.
- Then the code for this is static-branched out and never used again.
  - While not true in the past, the current random.c stays initialized forever once it is initialized. Entropy never “decreases” when used.
- Hopefully this is a conservative *underestimate*.
- Still, entropy estimation is fundamentally impossible.
  - And introduces a side channel attack: `/proc/sys/kernel/random/entropy_avail` leaks bits perhaps!
- Curious detail: `/dev/urandom` and `getrandom(GRND_INSECURE)` have to supply bytes before the RNG is initialized.
  - Attack: unprivileged userspace reads from /dev/urandom every time 1 bit of entropy is added, and bruteforces that bit. Doing this 256 times means the entire state is recovered.
  - Mitigation: add the first 128 bits as they come in – we want the most entropy as soon as possible during early boot – but buffer the next 128 bits.

Threat Models

- **Premature first**: We give output before accumulating enough entropy. Attacker can bruteforce state.
  - Do we care? *Yes.*
  - When the kernel boots, it is “compromised” by virtue of having an all zero state.
- **Premature next**: State of RNG leaks. We give output before accumulating enough new entropy after leak. Attacker can bruteforce state.
  - Do we care? *No.* This differs from Fortuna model, which doesn’t need to count entropy, but ensures that *eventually* the RNG state will recover from a compromise.
  - However, if the RNG state is compromised, it’s arguably more important to start using new entropy as soon as possible, to recover from that compromise, rather than the academic concern that it might never recover.
  - And of course one rarely even knows *when* a compromise happens. What’s important is that *if* it happens, there’s recovery.
  - Lack of care here goes against years of orthodoxy, but some recent academics seem to agree.
  - Side note: if the kernel RNG state is compromised, there are definitely worse problems to be concerned about.
    - How much do we care about scenarios where an attacker is only able to run the infoleak exploit just once, and not twice?

Threat Models Meet Reality: When to Reseed?

- Compromises might happen. When do we reseed? And how? That's the essential question.
- Our choice to prefer new entropy over old entropy as soon and as often as possible means the multiple pools of Fortuna aren't necessary.
- Currently we reseed every minute. Though we reseed every few seconds during the first minute of boot.
- This could be faster even. There are practical performance concerns too of course.
- Reseed after hibernation, after suspend, when a VM forks.
  - Reseeding always gets fresh RDRAND/RDSEED samples too, in addition to whatever else has accumulated in the entropy pool since the last reseed.

Kernel is King

- The kernel is in the best possible position for knowing *when* to reseed, based on various events.
- The kernel is in the best possible position to collect new entropic inputs.
- The set of heuristics and explicit designs that lead to deciding when to reseed and what to take as an entropic input is *the* evolutionary problem of the kernel RNG.

Diving In – Several Components

- Entropy collection
- Entropy extraction
- Entropy expansion
- Entropy sources
- “Are we initialized yet?”
- **Userspace interfaces**

Userspace Interfaces

- Generally wary of adding new ones. Current ones have created various pain points as the RNG has grown.
- `getrandom(0)` waits for initialization before giving out bytes. `getrandom(GRND_INSECURE)` does not.
- `/dev/random` waits for initialization before giving out bytes. `/dev/urandom` attempts to do jitter to get bytes, but if it fails, gives out insecure bytes.
  - I'd like to eventually unify these.
- All of these functions provide the **same bytes**. No difference in behavior *after initialization*.
- No signal interruption for `PAGE_SIZE` chunks, so you can always get at least 4k without `EINTR`.
- Recent discussions about vDSO acceleration for glibc.
- Documentation needs to be updated.

Inside the Linux Kernel RNG

- Recent modernization cleaned up the code, code documentation, and algorithms.
- Simplified design is cryptographically more rigorous.
- Code size is fairly minimal.
- Hopefully we're on the path toward solving boot time issues.
- Backported to all stable kernels: 4.9, 4.14, 4.19, 5.4, 5.10, 5.15.
- I'll be around all week – until Friday morning – and will also be milling around outside after this talk through lunch, so find me and let's talk.

Jason Donenfeld

- Personal website: www.zx2c4.com
- Company: www.edgesecurity.com
- Email: Jason@zx2c4.com


```
* This driver produces cryptographically secure pseudorandom data. It is divided
* into roughly six sections, each with a section header:
*
*   - Initialization and readiness waiting.
*   - Fast key erasure RNG, the "crng".
*   - Entropy accumulation and extraction routines.
*   - Entropy collection routines.
*   - Userspace reader/writer interfaces.
*   - Sysctl interface.
*
* The high level overview is that there is one input pool, into which
* various pieces of data are hashed. Prior to initialization, some of that
* data is then "credited" as having a certain number of bits of entropy.
* When enough bits of entropy are available, the hash is finalized and
* handed as a key to a stream cipher that expands it indefinitely for
* various consumers. This key is periodically refreshed as the various
* entropy collectors, described below, add data to the input pool.
```