7 Extending crypto-related backdoors to other scenarios

by BSDaemon and Pirata

This article expands on the ideas introduced by Taylor Hornby's "Prototyping an RDRAND Backdoor in Bochs" in PoC||GTFO 3:6. That article demonstrated the dangers of using instructions that generate a `#VMEXIT` event while in a guest virtual machine. Because a malicious VMM could compromise the randomness returned to a guest VM, it can affect the security of cryptographic operations.

In this article, we demonstrate that the newly available AES-NI instruction extensions in Intel platforms are vulnerable to a similar attack, with some additional badness. Not only guest VMs are vulnerable, but normal user-level/kernel-level applications that leverage the new instruction set are vulnerable as well, unless proper measures are in place. The reason for that is due to a mostly unknown feature of the platform, the ability to disable this instruction set.

7.1 Introduction

From Intel's website,[32]:

> Intel AES-NI is a new encryption instruction set that improves on the Advanced Encryption Standard (AES) algorithm and accelerates the encryption of data in the Intel Xeon processor family and the Intel Core processor family.

The instruction has been available since 2010.[33]

> Starting in 2010 with the Intel Core processor family based on the 32nm Intel micro-architecture, Intel introduced a set of new AES (Advanced Encryption Standard) instructions. This processor launch brought seven new instructions. As security is a crucial part of our computing lives, Intel has continued this trend and in 2012 and [sic] has launched the 3rd Generation Intel Core Processors, codenamed Ivy Bridge. Moving forward, 2014 Intel micro-architecture code name Broadwell will support the RDSEED instruction.

On a Linux box, a simple `grep` would tell if the instruction is supported in your machine.

```
bsdaemon@bsdaemon.org:~# grep aes /proc/cpuinfo
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm
constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc
aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3
cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave avx
f16c rdrand lahf_lm ida arat epb xsaveopt pln pts dtherm tpr_shadow vnmi
flexpriority ept vpid fsgsbase smep erms
```

A little-known fact, though, is that the instruction set can be disabled using an internal MSR on the processor. It came to our attention while we were looking at BIOS update issues and saw a post about a machine with AES-NI showing as disabled even though it was in, fact, supported.[34]

Researching the topic, we came across the MSR for a Broadwell Platform: `0x13C`. It will vary for each processor generation, but it is the same in Haswell and SandyBridge, according to our tests. Our machine had it locked.

```
MSR 0x13C
Bit     Description
0   Lock bit (always unlocked on boot time, BIOS sets it)
1   Not defined by default, 1 will disable AES-NI
2-32    Not sure what it does, not touched by our BIOS (probably reserved)
```

Discussing attack possibilities with a friend in another scenario—related to breaking a sandbox-like feature in the processor—we came to the idea of using it for a rootkit.

[32] `http://www.intel.com/content/www/us/en/architecture-and-technology/advanced-encryption-standard-aes-/data-protection-aes-general-technology.html`

[33] `https://software.intel.com/en-us/node/256280.`

[34] "AES-NI shows Disabled", `http://en.community.dell.com/support-forums/servers/f/956/t/19509653`

7.2 The Idea

All the code that we saw that supports AES-NI is basically about checking if it is supported by the processor, via CPUID, including the reference implementations on Intel's website. That's why we considered the possibility of manipulating encryption in applications by disabling the extension and emulating its expected results. Not long after we had that thought, we read in the PoC||GTFO 3:6 about `RDRAND`.

If the disable bit is set, the AES-NI instructions will return `#UD` (Invalid Opcode Exception) when issued. Since the code checks for the AES-NI support during initialization instead of before each call, winning the race is easy—it's a classic TOCTOU.

Some BIOSes will set the lock bit, thus hard-enabling the set. A write to the locked MSR then causes a general protection fault, so there are two possible approaches to dealing with this case.

First, we can set *both* the disable bit *and* the lock bit. The BIOS tries to enable the instruction, but that write is ignored. The BIOS tries to lock it, but it is ignored. That works unless the BIOS checks if the write to the MSR worked or not, which is usually not the case—in the BIOS we tested, the general protection fault handler for the BIOS just resumed execution. For beating the BIOS to this punch, one could explore the BIOS update feature, setting the `TOP_SWAP` bit, which let code execute *before* BIOS.[35] Chipsec toolkit[36] `TOP_SWAP` mechanism is locked.

For a Vulnerable Machine,

```
### BIOS VERSION 65CN90WW
OS      : uefi
Chipset:
VID:        8086
DID:        0154
Name:       Ivy Bridge (IVB)
Long Name: Ivy Bridge CPU / Panther Point PCH
[-] FAILED: BIOS Interface including Top Swap Mode is not locked
```

For a Protected Machine,

```
OS       : Linux 3.2.0-4-686-pae #1 SMP Debian 3.2.65-1+deb7u2 i686
Platform: 4th Generation Core Processor (Haswell U/Y)
         VID: 8086
         DID: 0A04
CHIPSEC : 1.1.7
[*] BIOS Top Swap mode is disabled
[*] BUC = 0x00000000 << Backed Up Control (RCBA + 0x3414)
  [00] TS                = 0 << Top Swap
[*] RTC version of TS = 0
[*] GCS = 0x00000021 << General Control and Status (RCBA + 0x3410)
  [00] BILD              = 1 << BIOS Interface Lock Down
  [10] BBS               = 0

[+] PASSED: BIOS Interface is locked (including Top Swap Mode)
```

The problem with this approach is that software has to check if the AES-NI is enabled or not, instead of just assuming the platform supports it.

Second, we can NOP-out the BIOS code that locks the MSR. That works if BIOS modification is possible on the platform, which is often the case. There are many options to reverse and patch your BIOS, but most involve either modifying the contents of the SPI Flash chip or single-stepping with a JTAG debugger.

Because the CoreBoot folks have had all the fun there is with SPI Flash, and because folk wisdom says that JTAG isn't feasible on Intel, we decided to throw folk wisdom out the window and go the JTAG route. We used the Intel JTAG debugger and an XDP 3 device. The algorithm used is provided in the attachment 3.

To be able to set this MSR, one needs Ring0 access, so this attack can be leveraged by a hypervisor against a guest virtual machine, similar to the `RDRAND` attack. But what's interesting in this case is that it can also be leveraged by a Ring0 application against a hypervisor, guest, or any host application! We used a Linux Kernel Module to intercept the `#UD`; a sample prototype of that module is in attachment 6.

[35] "Using SMM for other purposes", Phrack 65:7

[36] `https://github.com/chipsec/chipsec`

7.3 Checking your system

You can use the Chipsec module that comes with this article to check if your system has the MSR locked. Chipsec uses a kernel module that opens an interface (a device on Linux) for its user-mode component (Python code) to request info on different elements of the platform, such as MSRs. Obviously, a kernel module could do that directly. An example of such a module is provided with this article.

Since the MSR seems to change from system to system (and is not deeply documented by Intel itself), we recommend searching your OEM BIOS vendor forums to try and guess what is that MSR's number for your platform if the value mentioned here doesn't work. Disassembling your BIOS calls for the `wrmsr` might also help. Some BIOSes offer the possibility of disabling the AES-NI set in the BIOS menu, thus making it easier to identify the code (so dump the BIOS and diff). By default, the platform initializes with the disable bit unset, i.e., with AES-NI enabled. In our case, the BIOS vendor only set the lock bit.

7.4 Conclusion

This article demonstrates the need for checking the platform as whole for security issues. We showed that even "safe" software can be compromised, if the configuration of the platform's elements is wrong (or not ideal). Also note that forensics tools would likely fail to detect these kinds of attacks, since they typically depend on the platform's help to dissect software.

Acknowledgements

Neer Roggel for many excellent discussions on processor security and modern features, as well for the enlightening conversation about another attack based on disabling the AES-NI in the processor.

Attachment 1: Patch for Chipsec

This patch is for Chipsec (`https://github.com/chipsec/chipsec`) public repository version from March 9, 2015. A better (more complete) version of this patch will be incorporated into the public repository soon.

```
diff -rNup chipsec-master/source/tool/chipsec/cfg/hsw.xml chipsec-master.new/source/tool/chipsec/
    cfg/hsw.xml
--- chipsec-master/source/tool/chipsec/cfg/hsw.xml  2015-01-23 16:07:19.000000000 -0800
+++ chipsec-master.new/source/tool/chipsec/cfg/hsw.xml  2015-03-09 19:13:55.949498250 -0700
@@ -39,6 +39,10 @@
   <!--                                          -->
   <!-- ######################################## -->
   <registers>
+   <register name="IA32_AES_NI"    type="msr" msr="0x13c" desc="AES-NI Lock">
+      <field name="Lock"      bit="0"   size="1"  desc="AES-NI Lock Bit" />
+      <field name="AESDisable" bit="1" size="1" desc="AES-NI Disable Bit (set to disable)" />
+   </register>
   </registers>

-</configuration>
\ No newline at end of file
+</configuration>
diff -rNup chipsec-master/source/tool/chipsec/modules/hsw/aes_ni.py chipsec-master.new/source/tool
    /chipsec/modules/hsw/aes_ni.py
--- chipsec-master/source/tool/chipsec/modules/hsw/aes_ni.py  1969-12-31 16:00:00.000000000 -0800
+++ chipsec-master.new/source/tool/chipsec/modules/hsw/aes_ni.py  2015-03-09 19:22:12.693518998
    -0700
@@ -0,0 +1,68 @@
+#CHIPSEC: Platform Security Assessment Framework
+#Copyright (c) 2010-2015, Intel Corporation
+#
+#This program is free software; you can redistribute it and/or
+#modify it under the terms of the GNU General Public License
+#as published by the Free Software Foundation; Version 2.
+#
+#This program is distributed in the hope that it will be useful,
+#but WITHOUT ANY WARRANTY; without even the implied warranty of
+#MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
+#GNU General Public License for more details.
```

```
+#
+#You should have received a copy of the GNU General Public License
+#along with this program; if not, write to the Free Software
+#Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA.
+#
+#Contact information:
+#chipsec@intel.com
+#
+
+
+
+## \addtogroup modules
+# __chipsec/modules/hsw/aes_ni.py__ - checks for AES-NI lock
+#
+
+
+from chipsec.module_common import *
+from chipsec.hal.msr import *
+
+TAGS = [MTAG_BIOS,MTAG_HWCONFIG]
+
+class aes_ni(BaseModule):
+
+    def __init__(self):
+        BaseModule.__init__(self)
+
+    def is_supported(self):
+        return True
+
+    def check_aes_ni_supported(self):
+ return True
+
+    def check_aes_ni(self):
+        self.logger.start_test( "Checking if AES-NI lock bit is set" )
+
+        aes_msr = chipsec.chipset.read_register( self.cs, 'IA32_AES_NI' )
+        chipsec.chipset.print_register( self.cs, 'IA32_AES_NI', aes_msr )
+
+ aes_msr_lock =  aes_msr  & 0x1
+
+ # We don't really care if it is enabled or not since the software needs to
+ # test - the only security issue is if it is not locked
+ aes_msr_disable = aes_msr & 0x2
+
+ # Check if the lock is not set, then ERROR
+ if (not aes_msr_lock):
+   return False
+
+        return True
+
+    # ------------------------------------------------------------------------------
+    # run( module_argv )
+    # Required function: run here all tests from this module
+    # ------------------------------------------------------------------------------
+    def run( self, module_argv ):
+        return self.check_aes_ni()
```

Attachment 2: Kernel Module to check and set the AES-NI related MSRs

If for some reason you can't use Chipsec, this Linux kernel module reads the MSR and checks if the AES-NI lock bit is set.

```
#include <linux/module.h>
#include <linux/device.h>
#include <linux/highmem.h>
#include <linux/kallsyms.h>
#include <linux/tty.h>
#include <linux/ptrace.h>
#include <linux/version.h>
#include <linux/slab.h>
#include <asm/io.h>
#include "include/rop.h"
#include <linux/smp.h>
```

```
12
   #define _GNU_SOURCE
14
   #define FEATURE_CONFIG_MSR 0x13c
16
   MODULE_LICENSE("GPL");
18
   #define MASK_LOCK_SET              0x00000001
20 #define MASK_AES_ENABLED           0x00000002
   #define MASK_SET_LOCK              0x00000000
22
   void * read_msr_in_c(void * CPUInfo)
24 {
           int *pointer;
26         pointer=(int *) CPUInfo;
           asm volatile("rdmsr" : "=a"(pointer[0]), "=d"(pointer[3]) : "c"(FEATURE_CONFIG_MSR));
28         return NULL;
   }
30
   int __init
32 init_module (void)
   {
34         int CPUInfo[4]={-1};

36         printk(KERN_ALERT "AES-NI testing module\n");

38         read_msr_in_c(CPUInfo);

40         printk(KERN_ALERT "read: %d %d from MSR: 0x%x \n", CPUInfo[0], CPUInfo[3],
       FEATURE_CONFIG_MSR);

42         if (CPUInfo[0] & MASK_LOCK_SET)
                   printk(KERN_ALERT "MSR:  lock bit is set\n");
44
           if (!(CPUInfo[0] & MASK_AES_ENABLED))
46                 printk(KERN_ALERT "MSR:  AES_DISABLED bit is NOT set - AES-NI is ENABLED\n");

48         return 0;
   }
50
   void __exit
52 cleanup_module (void)
   {
54         printk(KERN_ALERT "AES-NI MSR unloading \n");
   }
```

Attachment 3: In-target-probe (ITP) algorithm

Since we used an interface available only to Intel employees and OEM partners, we decided to at least provide the algorithm behind what we did. We started with stopping the machine execution at the BIOS entrypoint. We then defined some functions to be used through our code.

```
1  get_eip():  Get the current RIP
   get_cs(): Get the current CS
3  get_ecx():  Get the current value of RCX
   get_opcode():  Get the current opcode (disassembly the current instruction)
5  find_wrmsr():  Uses the get_opcode() to compare with the '300f' (wrmsr opcode) and
     return True if found (False if not)
7  search_wrmsr():
     while find_wrmsr() == False:  step() -> go to the next instruction (single-step)
9  find_aes():
     while True:
11     step()
       search_wrmsr()
13     if get_ecx() == '0000013c':
         print "Found AES MSR"
15       break
```

Attachment 4: AES-NI Availability Test Code

This code uses the CPUID feature to see if AES-NI is available. If disabled, it will return "AES-NI Disabled". This is the reference code to be used by software during initialization to probe for the feature.

```c
#include <stdio.h>

#define cpuid(level, a, b, c, d) \
    asm("xchg{l}\t{%%}ebx, %1\n\t" \
        "cpuid\n\t" \
        "xchg{l}\t{%%}ebx, %1\n\t" \
        : "=a" (a), "=r" (b), "=c" (c), "=d" (d) \
        : "0" (level))

int main (int argc, char **argv) {
    unsigned int eax, ebx, ecx, edx;
    cpuid(1, eax, ebx, ecx, edx);
    if (ecx & (1<<25))
        printf("AES-NI Enabled\n");
    else
        printf("AES-NI Disabled\n");
    return 0;
}
```

Attachment 5: AES-NI Simple Assembly Code (to trigger the #UD)

This code will run normally (`exit(0)` call) if AES-NI is available and will cause a `#UD` if not.

```
Section .text
        global _start

_start:
    mov ebx, 0
    mov eax, 1
    aesenc xmm7, xmm1
    int 0x80
```

Attachment 6: #UD hooking

There are many ways to implement this, as 'Handling Interrupt Descriptor Table for fun and profit" in Phrack 59:4 shows. Another option, however, is to use Kprobes and hook the function `invalid_op()`.

```c
#include <linux/module.h>
#include <linux/kernel.h>

int index = 0;
module_param(index, int, 0);

#define GET_FULL_ISR(low, high) ( ((uint32_t)(low)) | (((uint32_t)(high)) << 16) )
#define GET_LOW_ISR(addr) ( (uint16_t)(((uint32_t)(addr)) & 0x0000FFFF) )
#define GET_HIGH_ISR(addr) ( (uint16_t)(((uint32_t)(addr)) >> 16) )

uint32_t original_handlers[256];
uint16_t old_gs, old_fs, old_es, old_ds;

typedef struct _idt_gate_desc {
        uint16_t offset_low;
        uint16_t segment_selector;
        uint8_t zero; // zero + reserved
        uint8_t flags;
        uint16_t offset_high;
} idt_gate_desc_t;
idt_gate_desc_t *gates[256];

void handler_implemented(void) {
        printk(KERN_EMERG "IDT Hooked Handler\n");
}

void foo(void) {
        __asm__("push %eax"); // placeholder for original handler

        __asm__("pushw %gs");
        __asm__("pushw %fs");
        __asm__("pushw %es");
        __asm__("pushw %ds");
        __asm__("push %eax");
```

```
        __asm__("push %ebp");
        __asm__("push %edi");
        __asm__("push %esi");
        __asm__("push %edx");
        __asm__("push %ecx");
        __asm__("push %ebx");

        __asm__("movw %0, %%ds" : : "m"(old_ds));
        __asm__("movw %0, %%es" : : "m"(old_es));
        __asm__("movw %0, %%fs" : : "m"(old_fs));
        __asm__("movw %0, %%gs" : : "m"(old_gs));

        handler_implemented();

        // place original handler in its placeholder
        __asm__("mov %0, %%eax": : "m"(original_handlers[index]));
    __asm__("mov %eax, 0x24(%esp)");

        __asm__("pop %ebx");
        __asm__("pop %ecx");
        __asm__("pop %edx");
        __asm__("pop %esi");
        __asm__("pop %edi");
        __asm__("pop %ebp");
        __asm__("pop %eax");
        __asm__("popw %ds");
        __asm__("popw %es");
        __asm__("popw %fs");
        __asm__("popw %gs");

        // ensures that "ret" will be the next instruction for the case
        // compiler adds more instructions in the epilogue
        __asm__("ret");
}

int init_module(void) {
        // IDTR
        unsigned char idtr[6];
        uint16_t idt_limit;
        uint32_t idt_base_addr;
        int i;

        __asm__("mov %%gs, %0": "=m"(old_gs));
        __asm__("mov %%fs, %0": "=m"(old_fs));
        __asm__("mov %%es, %0": "=m"(old_es));
        __asm__("mov %%ds, %0": "=m"(old_ds));

        __asm__("sidt %0": "=m" (idtr));
        idt_limit = *((uint16_t *)idtr);
        idt_base_addr = *((uint32_t *)&idtr[2]);
        printk("IDT Base Address: 0x%x, IDT Limit: 0x%x\n", idt_base_addr, idt_limit);

        gates[0] = (idt_gate_desc_t *)(idt_base_addr);
        for (i = 1; i < 256; i++)
                gates[i] = gates[i - 1] + 1;

        printk("int %d entry addr %x, seg sel %x, flags %x, offset %x\n", index, gates[index], (
    uint32_t)gates[index]->segment_selector, (uint32_t)gates[index]->flags, GET_FULL_ISR(gates[
    index]->offset_low, gates[index]->offset_high));

        for (i = 0; i < 256; i++)
                original_handlers[i] = GET_FULL_ISR(gates[i]->offset_low, gates[i]->offset_high);

        gates[index]->offset_low = GET_LOW_ISR(&foo);
        gates[index]->offset_high = GET_HIGH_ISR(&foo);

        return 0;
}

void cleanup_module(void) {
        printk("cleanup entry %d\n", index);

        gates[index]->offset_low = GET_LOW_ISR(original_handlers[index]);
        gates[index]->offset_high = GET_HIGH_ISR(original_handlers[index]);
}
```