Where am I? Operating System And Virtualization Identification Without System Calls

Jason L. Wright (jason@thought.net)
Cyber Security Symposium
April 17-19, 2017
Coeur d’Alene, Idaho, USA

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Distribution Statement A. Approved for public release. Distribution is unlimited.
What can I see? Given:

- **Code execution**
  - Arbitrary code
  - Unprivileged (user-level not kernel level)
  - Gather info without violating system call policy

- **It’s about knowing:**
  - what information is available
  - … and whether it is possible to monitor its access (or lie)
What is visible to user processes?

- Focus on x86 (32 and 64 bit, protected mode)
- General purpose registers:
  - EAX, EBX, ECX, EDX, EBP, ESP, ESI, EDI
- Normal segment selectors:
  - CS, DS, ES, FS, GS, SS
- Special segment selectors:
  - LDTR, TR
- Descriptor table registers:
  - GDTR, IDTR
- Machine Status Word: MSW
Segments: movl $1, (%eax)

- AT&T syntax (destination on the right)
- This means: move the constant 1 into the address stored in EAX
- In C: int *p = 1; // assuming p is stored in EAX

- The full address is DS:EAX
  DS is a segment selector (a 16 bit index into the OS controlled “Global Descriptor Table (GDT)” or “Local Descriptor Table (LDT)”
- Descriptor Table provides BASE, so virtual address is BASE+EAX
- Page tables turn BASE+EAX into the physical page of memory
So:

MOV $1, (%EAX)

Virtual address = DS:EAX

BASE = GDT[DS].BASE

Virtual address: DS:EAX = BASE + EAX

Physical address = page table[virtual address]
Reading current segments

Easy: PUSH CS; POP EAX

Push the current code segment onto the stack, pop it back into EAX

Works for all 6 “normal” segment selectors (CS, DS, ES, FS, GS, SS)

SLDT and STR instructions for fetching LDT and TR segment selectors
Probing for segments

LAR (load access rights)
LSL (load segment limit)
VERR (verify segment for reading)
VERW (verify segment for writing)

LAR and LSL allow probing for visibility of arbitrary segments
Can be used to generate “segment map” of the current environment

A list of valid segments, their type and their permissions
**Segment Example: FreeBSD 10.2 64bit**

<table>
<thead>
<tr>
<th>Segment</th>
<th>2</th>
<th>3</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>FS</td>
<td>FS</td>
<td>GS</td>
<td>CS</td>
<td>DS, ES, SS</td>
<td></td>
</tr>
<tr>
<td>Process local</td>
<td>Thread local</td>
<td>32bit code</td>
<td>data</td>
<td>64bit code</td>
<td></td>
</tr>
</tbody>
</table>

**Segment Example: Windows 10 64bit**

<table>
<thead>
<tr>
<th>Segment</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS, ES, GS, SS</td>
<td>CS</td>
<td>DS, ES, SS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32bit code</td>
<td>data</td>
<td>64bit code</td>
<td>Thread local</td>
<td></td>
</tr>
</tbody>
</table>
Machine Status Word (aka CR0)

“SMSW is only useful in operating-system software. However, it is not a privileged instruction and can be used in application programs. The [instruction] is provided for compatibility with the Intel 286 processor.”

Allows examination of Configuration Register 0 (CR0)
One bit of differentiation power: Alignment Mask bit

<table>
<thead>
<tr>
<th>AM bit</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear</td>
<td>Windows (XP, 8.1, 10 i386), MacOS 10.11, OpenBSD</td>
</tr>
<tr>
<td>set</td>
<td>Windows (10 amd64, Server 2016), Linux, FreeBSD</td>
</tr>
</tbody>
</table>
Descriptor Tables

“[SGDT, SIDT] [are] only useful in operating-system software. However, [they] can be used in application programs without causing an exception to be generated.”

Retrieve the base and limit of the GDT or IDT

(cannot access the contents of GDT/IDT, though)
<table>
<thead>
<tr>
<th>Base</th>
<th>Limit</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00020000</td>
<td>0xe00000</td>
<td>Linux 3.13 domU</td>
</tr>
<tr>
<td>0x8003f000</td>
<td>0x03ff</td>
<td>Windows XP SP3</td>
</tr>
<tr>
<td>0x80DCC000</td>
<td>0x03ff</td>
<td>Windows 8.1</td>
</tr>
<tr>
<td>0x8129C000</td>
<td>0x03ff</td>
<td>Windows 10</td>
</tr>
<tr>
<td>0xC15b07a4</td>
<td>0x0097</td>
<td>FreeBSD 10.3</td>
</tr>
<tr>
<td>0x15e10c00</td>
<td>0xffff</td>
<td>OpenBSD 5.8</td>
</tr>
<tr>
<td>0xf7beb000</td>
<td>0x00ff</td>
<td>Linux 2.6.13</td>
</tr>
<tr>
<td>0xf7be6000</td>
<td>0x00ff</td>
<td>Linux 3.13</td>
</tr>
<tr>
<td>0xf714f000</td>
<td>0x00ff</td>
<td>Linux 3.19</td>
</tr>
<tr>
<td>Base</td>
<td>Limit</td>
<td>OS</td>
</tr>
<tr>
<td>------------</td>
<td>----------</td>
<td>---------------</td>
</tr>
<tr>
<td>0x00000000</td>
<td>0x8169a450</td>
<td>FreeBSD 10.2</td>
</tr>
<tr>
<td>0xffff8000</td>
<td>0x003f</td>
<td>OpenBSD 5.8</td>
</tr>
<tr>
<td>0xffff8200</td>
<td>0x00ff</td>
<td>Linux 3.16 dom0</td>
</tr>
<tr>
<td>0xffff8200</td>
<td>0x00ff</td>
<td>Linux 3.16 domU</td>
</tr>
<tr>
<td>0xffff8800</td>
<td>0x00ff</td>
<td>Linux 3.16</td>
</tr>
<tr>
<td>0xffff8800</td>
<td>0x006f</td>
<td>Windows 10</td>
</tr>
<tr>
<td>0xffff8000</td>
<td>0x006f</td>
<td>Windows Server 2016</td>
</tr>
<tr>
<td>0xfffff800</td>
<td>0x0097</td>
<td>MacOS 10.11.6</td>
</tr>
</tbody>
</table>
Other sources of OS/Virtualization fingerprints

- **IDTR (like GDT identification)**
  - Linux fixmap IDT

- **CPUID**
  - Reserved leaf for hypervisors (EAX=0x40000000-4fffffff)
  - Hypervisor idiosyncrasies
    - Virtualbox RDRAND
    - Virtualbox XSAVE

- **FPU initialization**
  - FreeBSD 32bit initializes precision and reserved bit differently than everyone else
Future work

- Are there other sources of fingerprinting?
  - Are there mitigations?
- Would similar techniques work on other instruction sets?

Availability

- Utility for fetching data exposed to user processes:
  - https://github.com/wrigjl/ucpuinfo
- Modified Linux, Xen, and FreeBSD patches:
  - https://github.com/CyberGrandChallenge