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

Jason L. Wright
Thought Networks, LLC
525 Shoup Ave, Suite 319
Idaho Falls, Idaho 83402
jason@thought.net

ABSTRACT

Operating systems provide a wealth of versioning information via system calls, but suppose one is given control of the instruction pointer and allowed to jump to a provided block of code. Is it possible to identify the operating system and other key configuration details (bit width, virtualization environment, etc.) without resorting to system calls? We show that a host can be fingerprinted without tripping over the easiest and most commonly monitored behavioral characteristics of applications (system calls) and provide methods by which some the methods may be prevented.

CCS CONCEPTS

•**Software and its engineering** →**Virtual machines;** *Operating systems;* •**Security and privacy** →*Operating systems security; Virtualization and security;* •**Computer systems organization** →Processors and memory architectures;

KEYWORDS

virtual machine, operating system, fingerprinting, identification

ACM Reference format:
Jason L. Wright. 2017. Where am I? Operating System and Virtualization Identification Without System Calls. In *Proceedings of Cybersecurity Symposium, Coeur d'Alene Idaho USA, April 2017 (CSS'17),* 4 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Host fingerprinting is the art of identifying unique aspects of a host using intended or unintended indications from the host. Attackers writing targeted exploit software need accurate fingerprints to hone their attacks. Some fingerprinting methods work remotely such as TCP/IP stack fingerprinting [4] and Passive OS Fingerprinting [16]. Other methods use operating system queries such as *uname()* under UNIX-like operating systems or *GetSystemInfo()* on Microsoft Windows. The latter methods require code execution and use system calls which are easily monitored and intercepted by security applications such as *strace*, SECCOMP-BPF [3], or Process Monitor [14].

With the ability to execute arbitrary code on a host, we show that it is possible to fingerprint a host without using system calls on Intel IA-32 (x86) and X86-64 (amd64) machines using intrinsic characteristics of the processor and operating system. In some cases, the virtualization environment is distinguishable using similar methods.

The rest of this paper is organized as follows: Section 2 provides the background, Section 3 demonstrates the method for fingerprinting without system calls, Section 4 discusses mitigation methods where possible, Section 5 discusses related work, and Section 6 provides the conclusion and directions for future work.

2 BACKGROUND

All memory accesses (instruction and data) on IA-32 are implicitly or explicitly made through segments. Instructions such as "mov $1, (%edx)" which moves the literal "1" into the address pointed to by the EDX register and the data segment selector DS (implicit). Segments define the permissions on the memory region (readable, writeable, and/or executable), the base address, length, and what privilege level is required to access the memory region (e.g. kernel or user-space). If a memory address is greater than the defined length of the segment, the process generates a general protection (GP) fault. The base address of the segment is added to the register to form the canonical address which is used to complete the access. Page tables and virtualization determine the physical address finally accessed.

IA-32 defines a set of eight segment selectors. Code Segment (CS) is implicitly used for all references to executable memory. Stack Segment (SS) is used for accesses involving the stack pointer register (ESP). Data Segment (DS) is used implicitly by most memory operations. Extra Segment (ES) is used implicitly by some memory copy operations. Two additional segment selectors FS and GS are available for use by the operating system. Finally, two other segment selectors are exclusively used by the operating system: LDTR (local descriptor table) and TR (Task Register).

Each segment selector references an entry in the Global Descriptor Table (GDT). In a modern operating system, the GDT is maintained by the operating system. The GDT entry for most segments (CS, DS, ES, FS, GS, and SS) refers to a code or data segment tuple (base address, permissions, and length). The Local Descriptor Table Register and Task Register reference specialized entries in the GDT.

For the purposes of this paper, we assume we can execute arbitrary user-space code. Access to kernel data structures is not required for this discussion. Put in terms of IA-32, the code described runs at Current Privilege Level 3 (CPL3). Normally, operating system kernel code runs at with the highest privileges (lowest CPL) known as CPL0.

3 PROBING FOR FINGERPRINTS

Given arbitrary unprivileged IA-32 code can be executed, the data visible to the program should be enumerated. This amounts to the instantaneous state of a program: all of its general registers, instruction pointer, and memory contents. For IA-32 an abbreviated process state follows.

- 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

For fingerprinting the general purpose registers are of limited value. If the process is single threaded and relatively simple, a stack pointer below 0x00ffffff is a weak indicator of Microsoft Windows. UNIX-like operating systems typically have stack pointers set to the highest reaches of user-space memory (greater than 0x80000000). The general purpose nature of these registers means that there are no strong guarantees that they are used for any particular purpose at a given instance. For fingerprinting, we must look elsewhere.

3.1 Segments

The normal segment selectors reference entries in the GDT or LDT, and the instructions LAR (Load Access Rights), LSL (Load Segment Limit), VERR (Verify Segment for Reading), and VERW (Verify Segment for Writing), allow for probing the existence and configuration of a segment. It is also possible to probe for the existence of segments by iterating over all possible segments. Segment selectors are 16bit registers, but 1 bit is used to indicate whether the segment is in the GDT or LDT, and 2 bits indicate the privilege level of the segment; this leaves 8192 possible segments. Operating systems typically expose a small number (less than 10) segments to user processes. The minimal set capable of executing a program is an executable segment (*CS*) and a data segment (*DS* = *ES* = *SS*).

For 32-bit operating systems wishing to support the SYSENTER instruction, the following descriptors must be contiguous and in order: kernel code, kernel data, 32-bit user code, and user data. For 64-bit operating systems supporting the SYSCALL instruction, adding a 64-bit code segment after the user data segment is sufficient to support all methods of supporting system calls while allowing for efficient execution of 32- and 64-bit system calls. Kernel segments are not normally visible to user processes.

Table 1 shows the segments visible to user programs for various operating systems and configurations. *i386* refers to a 32-bit operating system; *amd64* is a 64-bit operating system; *domU* is a paravirtualized operating system running under Xen 4.4; *dom0* is the host for Xen virtual machines.

Using the visible segments from Table 1, it is possible to differentiate major operating systems: Microsoft Windows from Linux, from FreeBSD, from OpenBSD, etc. Using segments alone will not differentiate OpenBSD from MacOS. Further, the presence of paravirtualization provided by the Xen hypervisor is identified by presence of the low (1–2) and high (7172–7174) numbered segments.

OS	visible segments
Windows XP SP3 i386 Windows 8.1 i386	3 (32), 4 (data), 7, 8
Windows 10 i386	3 (32), 4 (data), 7, 8, 12
Windows 2016 amd64 Windows 10 amd64	4 (32), 5 (data), 6 (64), 10
Linux 2.6.13 i386 Linux 3.13 i386	6, 14 (32), 15 (data)
Linux 3.13 i386 domU	6, 14 (32), 15 (data), 7173 (32), 7174 (data)
Linux 3.16 amd64	4 (32), 5 (data), 6 (64), 15
Linux 3.13 amd64 domU	1, 2, 3, 4, 5, 6, 15, 7172 (32), 7173 (data), 7174 (64)
Linux 3.16 amd64 dom0	1, 2, 3, 4, 5, 6, 7172 (32), 7173 (data), 7174 (64)
FreeBSD 10.3 i386	3, 6 (32), 7 (data), 10
FreeBSD 10.3 amd64	2, 3, 6 (32), 7, 8 (64)
OpenBSD 5.8 i386	5 (32), 6 (data), 11, 13
OpenBSD 5.8 amd64 MacOS 10.11.6 amd64	3 (32), 4 (data), 5 (64)

Table 1: Segments visible to user processes (32 is a 32-bit code segment, data is a data segment, and 64 is a 64-bit code segment)

Table 2: Alignment Mask differentiation using SMSW

AM bit	Operating Systems
clear	Windows (XP, 8.1, 10 i386), MacOS 10.11.6, OpenBSD
set	Windows (10 amd64, Server 2016, Linux, FreeBSD

3.2 Machine Status Word

The IA-32 instruction set provides the SMSW (Store Machine Status Word) instruction which returns the value of Configuration Register 0 (CR0). Modern processors provide the "MOV CR*n*" instructions for read and writing the configuration registers; this family of instructions is privileged and thus unavailable to user processes. According to Intel:

> *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[7, p. 4-367].*

The only bit in CR0 that provides any ability to differentiate operating systems is the Alignment Mask (AM) bit. Table 2 shows the differentiation provided by this bit.

3.3 Descriptor Tables

The IA-32 instruction set provides four instructions for retrieving information about descriptor tables: SGDT (Store Global Descriptor Table Register), SIDT (Store Interrupt Descriptor Table Register), SLDT (Store Local Descriptor Table Register), and STR (Store Task

Table 3: Differentiation based on GDT

GDT base	GDT limit	OS
32-bit Operating System		
`0x00.020000`	`0xefff`	Linux 3.13 domU
`0x80.03f000`	`0x03ff`	Windows XP SP3
`0x80.dcc000`	`0x03ff`	Windows 8.1
`0x81.29c000`	`0x03ff`	Windows 10
`0xc1.5b07a4`	`0x0097`	FreeBSD 10.3
`0xf4.e1c000`	`0xffff`	OpenBSD 5.8
`0xf7.beb000`	`0x00ff`	Linux 2.6.13
`0xf7.be6000`	`0x00ff`	Linux 3.13
`0xf7.14f000`	`0x00ff`	Linux 3.19
64-bit Operating System		
GDT base	**GDT limit**	**OS**
`0x000000008169a450`	`0x0067`	FreeBSD 10.2
`0xffff800000011068`	`0x003f`	OpenBSD 5.8
`0xffff820000000000`	`0xefff`	Linux 3.16 dom0
`0xffff820000020000`	`0xefff`	Linux 3.16 domU
`0xffff88002fc09000`	`0x007f`	Linux 3.16
`0xfffff80028dc3000`	`0x006f`	Windows 10
`0xfffff80089a54000`	`0x006f`	Windows Server 2016
`0xffffff8000001000`	`0x0097`	MacOS 10.11.6

Register). All four of these instructions are available to user processes (no elevated privilege required). SGDT and SIDT return the address of the global and interrupt descriptor tables respectively. SLDT and STR return a segment number for an entry in the GDT where information can found about the local descriptor table and task register, respectively.

The address of the GDT or IDT is of little use to user processes since the memory referenced is not mapped into user processes. According to Intel:

> *(SGDT [7, p. 4-349], SIDT [7, p. 4-363]) is useful only by operating-system software. However, it can be used in application programs without causing an exception to be generated.*

For fingerprinting, the base and limit provided by the SGDT instruction provides a strong indication of operating system as shown in Table 3. Using the most significant 8-bits of the base easily differentiates 32-bit operating system families. The most significant bits can also be used to identify 64-bit operating systems. The GDT limit value differentiates the major operating system families (and identifies the use of the paravirtualization of the Xen hypervisor). It is interesting to note that neither FreeBSD 10.2 nor OpenBSD 5.8 align the GDT on a page boundary (multiple of 4096 bytes) for their amd64 ports.

3.4 CPUID

The CPUID instruction was added to IA-32 with the i486SL and Pentium generation of processors. It is an unprivileged instruction which returns configuration and feature availability information about a processor core. Operating systems cannot intercept the use of CPUID by user processes; there is neither an interrupt nor an exception generated which would allow for it.

Table 4: CPUID virtualization leaf 0x40000000 strings

Product	String
VMWare	`VMwareVMware`
Xen	`XenVMMXenVMM`
VirtualBox	`KVMKVMKVM`

The CPUID leaves from 0x40000000 to 0x4fffffff have been reserved and this range is now used by virtual machine monitors to allow guests to query virtualization information. Table 4 shows strings returned by various virtualization products in the CPUID virtualization leaf. This behavior is intended and can be overridden in each virtualization product by specifying the values to be returned for CPUID queries.

Virtualization products also change other CPUID leaves in other detectable ways. For instance, all modern Intel processors support the RDRAND instruction for generating random numbers, but VirtualBox indicates that the instruction is not supported. VirtualBox 4 and VirtualBox 5 can be differentiated by examining the state of XSAVE bit in CPUID leaf 1 because VirtualBox 4 does not support the XSAVE family of instructions supported on all modern Intel processors.

4 DISCUSSION

The sensitivity of the instructions in Section 3 was first described in [13]. At the time of its writing there was no method by which the SGDT, SIDT, STR, SLDT instructions could be intercepted. In fact, several techniques for identifying whether virtualization is in use have been proposed using subsets of these instructions [9–12, 15]. Processors based on Intel Nehalem and Westmere processors running hypervisors can intercept these sensitive yet unprivileged instructions using *descriptor table exiting* [6] where the hypervisor gets control of the processor when one of these instructions is executed by a guest. The hypervisor can return arbitrary values to the guest. The author has provided proof of concept patches against the Xen hypervisor (see Section 8).

Likewise, the SMSW instruction (Section 3.2) is equivalent to a move from a control register. Access to control registers can be intercepted by a hypervisor which can determine whether to return the true value or substitute an arbitrary value. Additionally, using hypervisor mask and shadow registers [8, pp. 24:11–24:12], a hypervisor can ensure that reserved bits in CR0 always contain arbitrary values. This is also implemented in the proof of concept patch for Xen (Section 8).

Preventing identification based on the exposed segments (Section 3.1) is more difficult than the descriptor table and machine status word identification described previously because there is no hypervisor handling for the VERR, VERW, LSL and LAR instructions. Instead, the easiest way to prevent OS/Virtualization detection using these instructions is to mimic the segment configuration of another operating system. As described in Section 3.1, a few restrictions are placed on the ordering of segments so that operating systems can support various system call methods (INT $0x80, SYSCALL / SYSRET, SYSENTER / SYSEXIT). As long as the segments are contiguous and in the proper order, a 32-bit user code segment can be any number $3 <= N <= 65533$. The local descriptor table can be made invisible to processes not using it.

MacOS and OpenBSD amd64 expose the minimal set of segments (32-bit code, data, and 64-bit code) required to support single-threaded 32- and 64-bit binaries simultaneously (Table 1). Linux/amd64 adds one additional segment (15) that is used to support thread-local storage. As a proof of concept, the author has modified FreeBSD/amd64 to have the Linux/i386 descriptor table configuration when running 32-bit binaries and Linux/amd64 when running 64-bit binaries. Primarily this is a matter of rearranging the descriptor table to match the usage in Linux. Additionally, the when switching contexts, the LDT is marked as "not present" in the GDT.

Disguising CPUID and being self-consistent is difficult. However, it is possible to make one virtualization environment indistinguishable from another. Fortunately, hypervisors are unconditionally called to interpret CPUID requests. Using a modified hypervisor it is possible to examine requests, intercept, and modify them without user or even guest kernel knowledge. To demonstrate this, the the Xen hypervisor was made to resemble VirtualBox 4 by intercepting the VMExit (trap from guest operating system to hypervisor) for CPUID and returning the values that would normally be returned by VirtualBox in response to queries.

5 RELATED WORK

Many techniques have been developed for networked system identification [4, 16]. The fingerprinting techniques described in this paper require code execution on the system to be fingerprinted. We rely on the ability to execute code native to the IA-32 and/or X86-64 instruction sets without using system calls.

Standards such as PECOFF and ELF define the register usage and calling conventions required to implement an Application Binary Interface (ABI); additional processor features are left unspecified. The execution environment required by conventional ABIs are a subset of the features actually exposed by modern processors. The techniques described in this paper exist in difference in these two sets. Various tools have been proposed to further restrict or monitor the operating system interface between an application and the kernel [3, 5, 14], but these tools do not proclaim to restrict processor features.

QEMU user mode emulation [2] provides the system call interface and ABI of a target operating system while running on a host of a potentially different architecture. As such, it emulates the execution of the sensitive, yet unprivileged instructions described in this paper. Dune [1] provides user processes with access to privileged instructions using the virtualization techniques similar to those described in Section 4.

6 CONCLUSION

We have demonstrated that it is possible to fingerprint the operating system and virtualization product without resorting to system calls. In some cases it is possible to mitigate the exposed information, however in the case of certain instructions, no possible mitigation is available except by using a hypervisor. The author has made available a modified hypervisor and operating system patches which demonstrate how fingerprinting can be mitigated.

IA-32 has a rich ecosystem of operating systems and virtualization environments. An interesting research question is whether similarly rich environments such as those built around modern ARM processors provide similar fingerprinting opportunities. It is also possible that processors of more recent design expose fewer operating system intrinsics to user processes.

7 ACKNOWLEDGEMENTS

This work was sponsored by the Defense Advanced Research Projects Agency under Air Force Contract # FA8750-12-D-0005. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

The author would also like to thank the anonymous reviewers for their time and feedback.

8 AVAILABILITY

An open source program for probing the exposed segments, CPUID, GDTR, LDTR, etc. is available on the author's GitHub:

`https://github.com/wrigjl/ucpuinfo`

Modified Linux, Xen, and FreeBSD patches implementing the mitigations in Section 4 are available on GitHub:

`https://github.com/CyberGrandChallenge`

REFERENCES

[1] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX, Hollywood, CA, 335–348. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/belay
[2] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX Annual Technical Conference, FREENIX Track*. 41–46.
[3] Will Drewry. 2012. Seccomp filtering. Linux Kernel documentation. (Jan 2012). https://lwn.net/Articles/475049/
[4] Fyodor. 1998. Remote OS detection via TCP/IP stack fingerprinting. (October 1998). http://www.insecure.org/nmap/nmap-fingerprinting-article.html
[5] Poul henning Kamp and Robert N. M. Watson. 2000. Jails: Confining the omnipotent root. In *In Proc. 2nd Intl. SANE Conference*.
[6] Intel 2009. *A Primer On Virtualization*. Intel. https://software.intel.com/sites/default/files/m/1/d/5/7/f/26677-A-Primer-on-Virtualization.pptx
[7] Intel. 2014. *Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 2: Instruction Set Reference*.
[8] Intel. 2014. *Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 3: System Programming Guide*.
[9] Tobias Klein. 2003. jerry – A(nother) VMware Fingerprinter. (2003). https://web.archive.org/web/20070827111813/http://www.trapkit.de/research/vmm/jerry/jerry.c
[10] Tobias Klein. 2008. ScoopyNG - The VMware detection tool. (2008). http://www.trapkit.de/research/vmm/scoopyng/index.html
[11] Danny Quist and Val Smith. *Detecting the Presence of Virtual Machines Using the Local Data Table*. Technical Report. Offensive Computing.
[12] Danny Quist and Val Smith. 2006. Further Down the VM Spiral: Detection of full and partial emulation for IA-32 virtual machines. In *DEFCON 14*. https://dl.packetstormsecurity.net/papers/general/dquist_valsmith_further_down_the_vm_spiral.pdf
[13] John Scott Robin and Cynthia E. Irvine. 2000. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *9th Annual Security Symposium*. USENIX, 129–144.
[14] Mark Russinovich. 2016. Windows Sysinternals: Process Monitor. (2016). https://technet.microsoft.com/en-us/sysinternals/processmonitor.aspx
[15] Joanna Rutkowska. 2004. Red Pill... or how to detect VMM using (almost) one CPU instruction. (November 2004). https://web.archive.org/web/20041130172213/http://invisiblethings.org/papers/redpill.html
[16] Michal Zalewski. 2000. p0f — passive os fingerprinting. (June 2000). http://seclists.org/bugtraq/2000/Jun/141