Using VMX within Linux

We explore the feasibility of executing ROM-BIOS code within the Linux x86_64 kernel

Motivation

- Versions of Linux for the Intel x86 platform previously have provided a system-call to allow execution of ‘real-mode’ ROM-BIOS routines in the CPU’s ‘Virtual-8086’ mode
- For example, an application (with suitable privileges) could easily invoke a standard SVGA BIOS routine that would modify the graphics display modes for special effects

EM64T

- But support for ‘Virtual-8086’ emulation is not implemented within the CPU’s IA-32e “compatibility mode”, compelling removal of the Linux ‘vm86()’ system-call from the kernel-versions for x86_64 platforms
- So we wanted to know if ‘Virtualization Technology’ could offer a way of restoring ‘Virtual-8086’ emulation under 64-bit Linux

How it worked

- Applications that wanted to execute ‘real-mode’ ROM-BIOS routines could setup a data-structure (‘struct vm86plus_struct’) having entries for the CPU’s registers, then pass that structure to the kernel for execution of the real-mode code, using the supplied register-values as parameters to modify the underlying hardware settings
- Let’s try to mimic that idea in 64-bit Linux

The sixteen CPU registers

General-purpose registers

EAX
ECX
EDX
EBX

Pointer and Index registers

ESP
EBP
ESI
EDI

Segment/selector registers

ES
CS
SS
DS
FS
GS

Control and Status 'Flags' register

EFLAGS

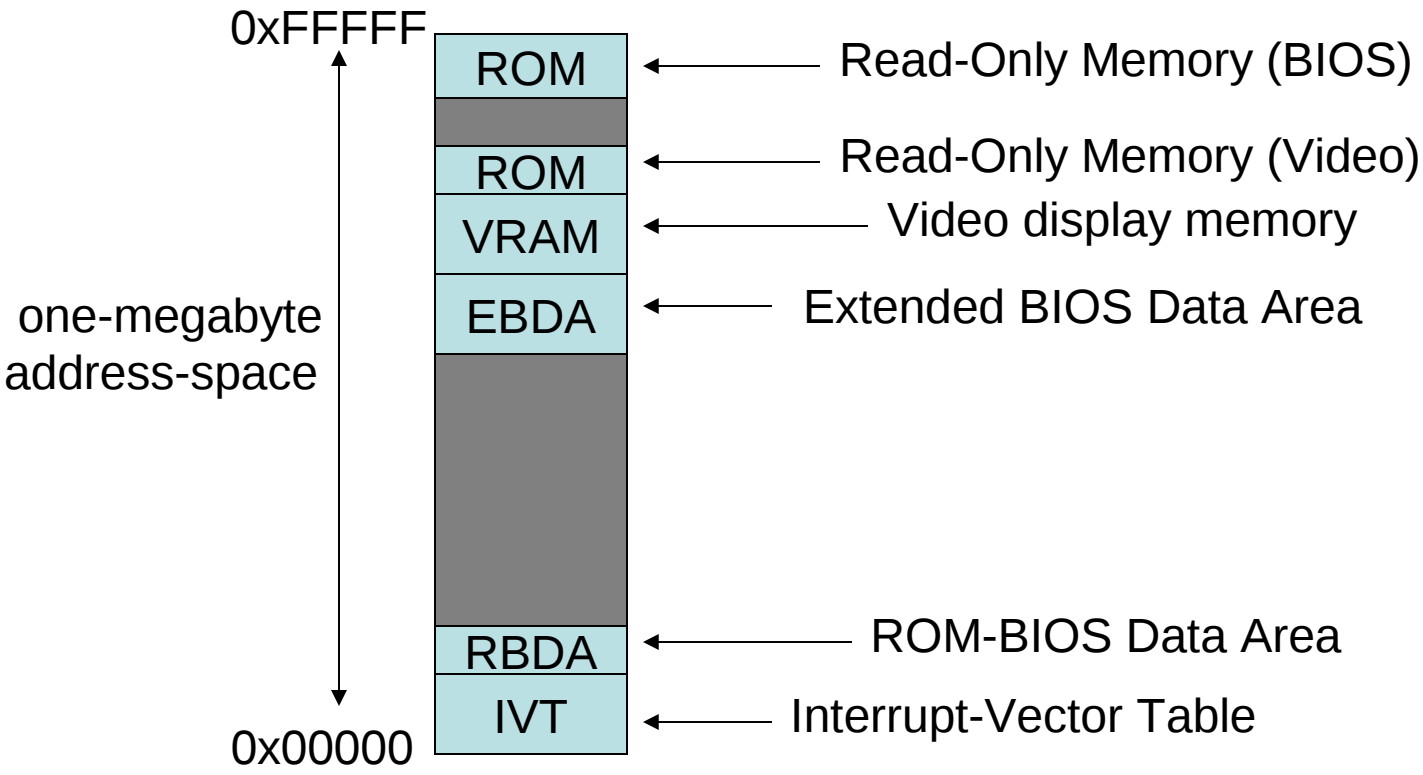
Instruction-Pointer register

EIP

We create a 'struct' that has fields for all sixteen of these registers ('myvmx.h')

The ‘real-mode’ address-space

- Code that uses ‘real-mode’ addresses is limited to the bottom megabyte of memory



Example: INT-0x12

- ROM-BIOS includes an interrupt-service routine for ‘Get_Memory_Size’ function (invoked when ‘int $0x12’ executes in real-mode); it returns a value in register AX giving the amount of available real-mode memory (expressed in kilobytes)
- If the (optional) EBDA is not present, the value returned in AX would be 0x0280 (=512+128 = 640 KB)

‘tryoutpc.cpp’

- Our demo-program will attempt to execute this ‘INT $0x11’ interrupt service routine because, since it returns a value in AX, we will be able to see whether or not it worked
- Our Core-2 Duo machines don’t have any display-monitors attached, so couldn’t see if graphics display-modes did get changed by executing the Video BIOS routines

Installable kernel modules

- The VMX instructions that setup a VM and a VMM must execute with ring0 privileges, so we need to add such code to the Linux kernel – either by modifying the kernel and then recompiling and reinstalling (UGH!!) or, more conveniently, by ‘installing’ the additional code in a ‘module’ at runtime
- If ‘bugs’ are present, we can remove our module, fix it, and then easily reinstall it

Module-format

- Two required ‘management’ functions:
  - for ‘initializations’ at installation-time
  - for ‘cleanup’ at removal-time
- Additional ‘payload’ functions:
  - to provide kernel-services to applications via standard UNIX programming interfaces
- Module code-license:
  - to cope with legal issues (e.g., copyrights)

‘linuxvmm.c’

- Our demo will be a character-mode Linux device-driver, accessible via a ‘device-file’ (named ‘/dev/vmm’) which an application can ‘open()’ with a call to standard C/C++ library-functions
- It will provide two service-functions:
  - memmap()
  - ioctl()

‘mmap()’

- Calling this function will request the kernel to create entries in a task’s page-mapping tables that give it a one-megabyte virtual address-space, initialized in the way that real-mode ROM-BIOS code would expect

‘ioctl()’

- Calling this function, with a ‘struct’ that has fields for the sixteen program-registers, asks the kernel to create a Host VMM that will continue to execute the kernel’s code in 64-bit mode, and to launch a Guest VM that will execute the ‘real-mode’ procedure in Virtual-8086 mode, using the sixteen parameter-values in the VM’s registers, returning modifications upon a VM-exit

‘init_module()’

- The mandatory ‘init_module()’ function will verify that the CPU supports Intel’s VMX instructions, will allocate kernel-memory for the required VMCS structures and for the guest-task’s various data-structures, and will read various system-registers (such as the VMX-Capability MSRs)
- And it will ‘register’ the driver’s functions

‘cleanup_module()’

- This module-function will ‘unregister’ the driver’s functions, and free the previously allocated kernel-memory, when a user is finished with the device-driver’s services

Compiling the module

- Special actions are needed for compiling modules for Linux kernel-versions 2.6.x
- We’ve provided a tool that automates this step for you (named ‘mmake.cpp’); use it like this:

```
$  mmake  linuxvmm
```

‘/dev/vmm’

- Special privileges are needed by a user to execute the Linux commands that will set up this device ‘special’ file (it has already been setup by our System Administrator on the classroom and CS lab machines)
- If you want to do it on your own machine, do this (as ‘root’ user):

```
root#  mknod  /dev/vmm  c  88  0
root#  chmod  a+rw  /dev/vmm
```

In-class exercise #1

- Try downloading and compiling the demo-program (‘tryoutpc.cpp’) and the module (‘linuxvmm.c’), then install the module on your ‘anchor’ machine and run the demo

In-class exercise #2

- After you have run the ‘tryoutpc’ program, install the ‘dram.c’ device-driver and use our ‘fileview’ utility to examine the memory-areas where Host and Guest VMCS data-structures were located
- You can discover the physical address of the Host’s VMCS by using the command:

  $ cat /proc/linuxvmm

- The following page-frame is the Guest’s VMCS

Instructor’s Addendum

- Our initial version of ‘tryoutpc.cpp’ tried to execute the ROM-BIOS interrupt-handler for ‘int-0x12’ (Get_Memory_Size), which worked quite nicely on our Pentium-D development-platform at home -- but not on our Core-2 Duo machines at campus!
- So we’ve substituted the ‘int-0x11’ handler (Equipment_Check) in place of ‘int-0x12’ in order to have a nice working demo