CLOUDBURST

A VMware Guest to Host Escape Story

Kostya Kortchinsky

Immunity, Inc.

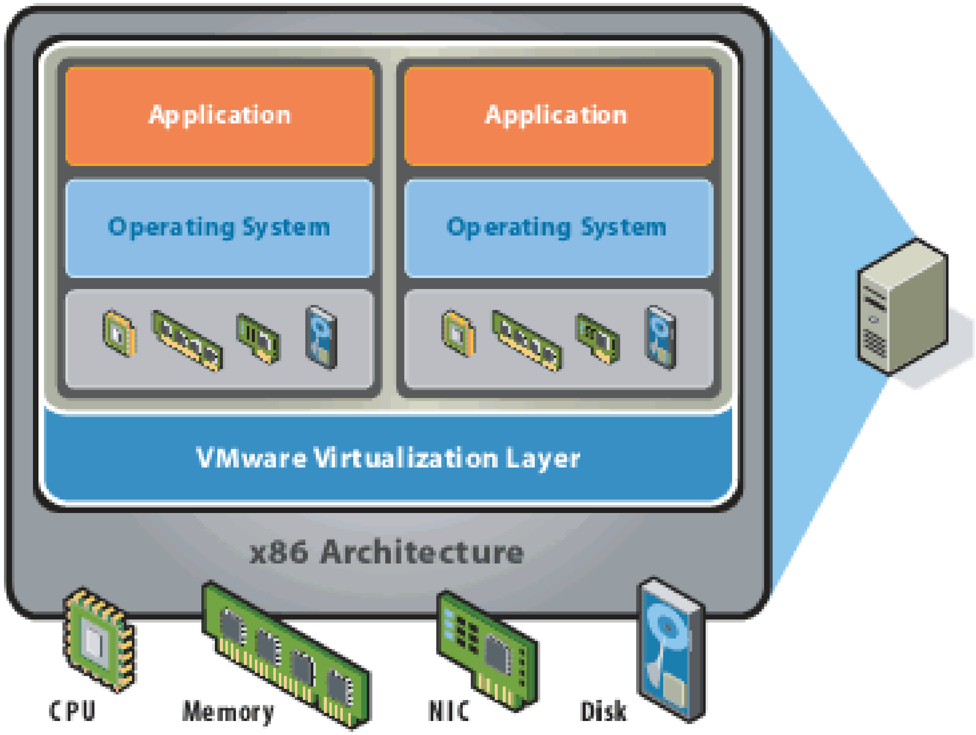
主要是说怎么样通过设备模拟进行虚拟机逃逸

BlackHat USA 2009, Las Vegas

Introduction

VMware Architecture

KNOWING YOU'RE SECURE



Devices are emulated on the Host

Why devices?

KNOWING YOU'RE SECURE

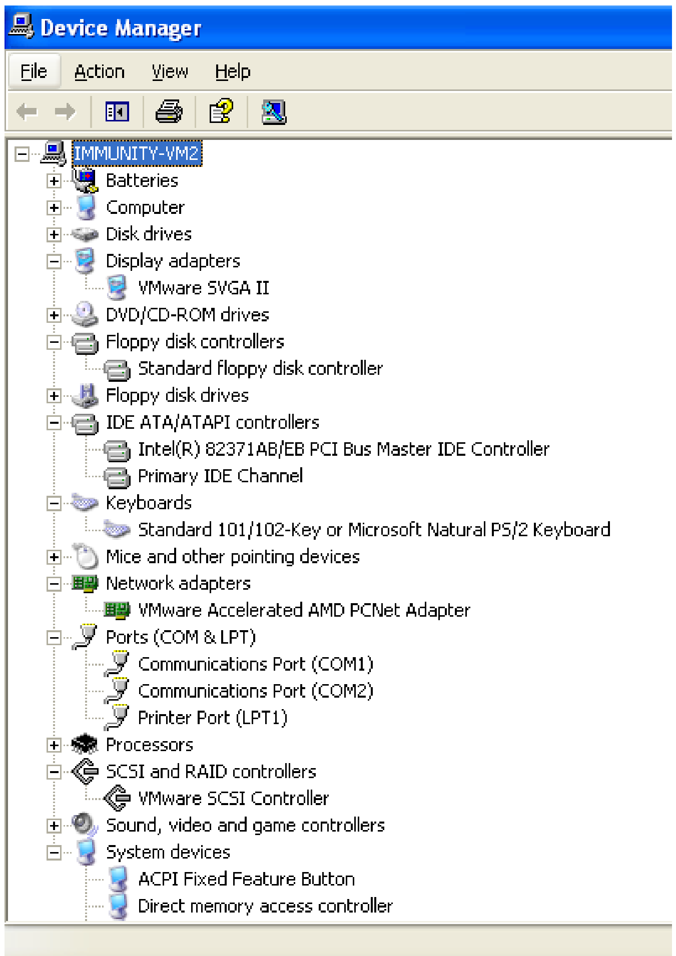
- I don't have enough low-level system Mojo ☹
- They are common to all VMware products
- They “run” on the Host
  - vmware-vmx process

设备的模拟在vmware-vmx进程上

- They can be accessed from the guest
  - Through Port I/O or memory-mapped I/O
- They are written in C/C++
- They sometimes parse some complex data!

Devices on a VM

KNOWING YOU'RE SECURE



Windows XP SP3 (ESX)

1. **Video adapter**
2. Floppy controller
3. IDE controller
4. Keyboard controller
5. Network Adapter
6. COM/LPT controller
7. SCSI controller(s)
8. DMA controller
9. ~~USB controller (WKS)~~
10. ~~Audio adapter (WKS)~~

CLOUDBURST

KNOWING YOU'RE SECURE

- Combination of 3/4 bugs in the VMware emulated video device
  - Host memory leak into the Guest
  - Host arbitrary memory write from the Guest
    - Relative
    - Absolute
  - And some additional DEP friendly goodness
- Reliable Guest to Host escape on recent VMware products: Workstation, Fusion?, ESX Server (4.0 RC Hardfreeze)

VMware SVGA II

VMware Publications

KNOWING YOU'RE SECURE

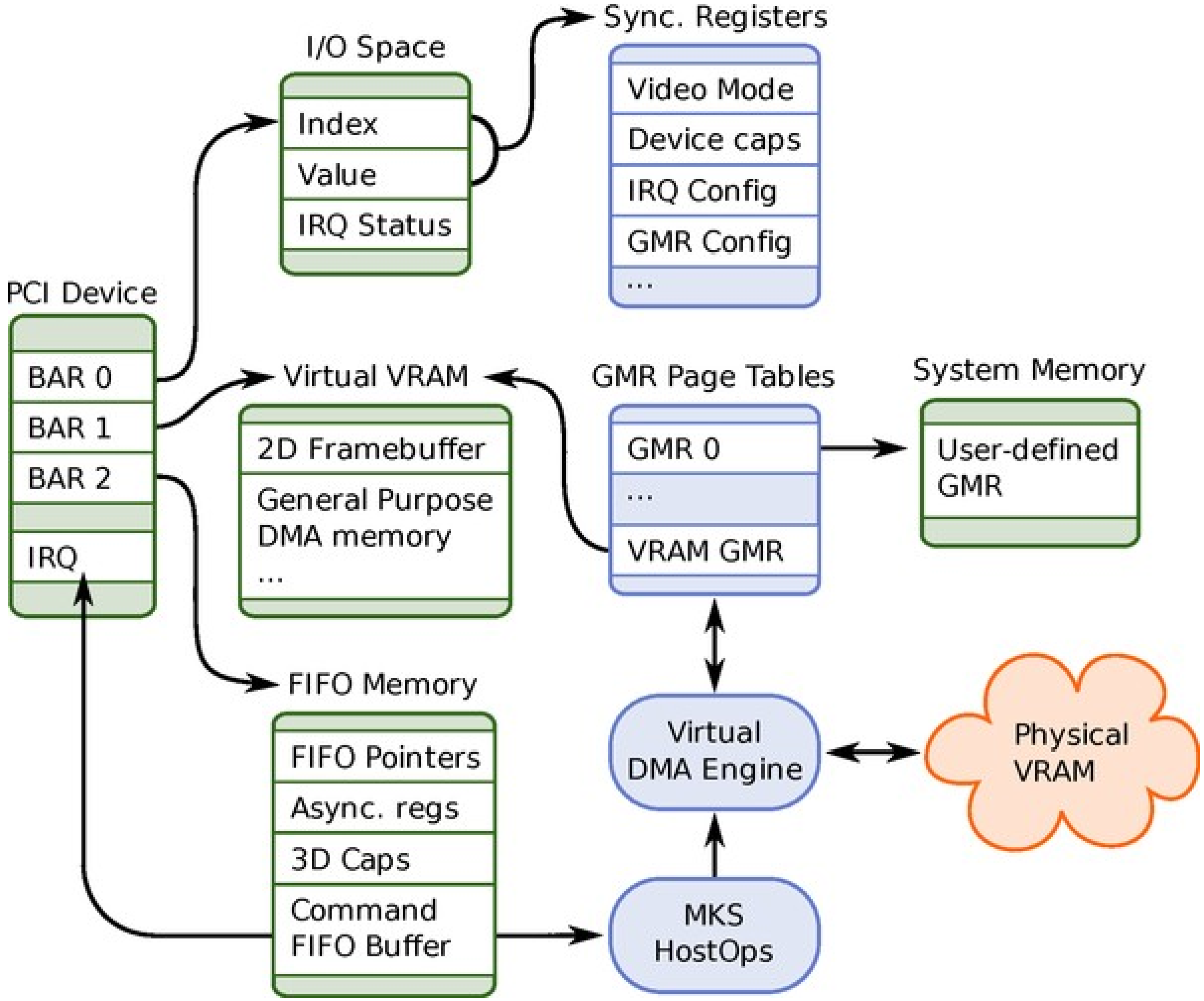
- GPU Virtualization on VMware's Hosted I/O Architecture by Micah Dowty, Jeremy Sugerman
  - We were not aware of this paper during our research
  - Good insight on the technology
- Previous VMware security announcements have included device driver guest->host vulnerabilities, as have Microsoft VirtualServer and Xen
- I am not a virtualization specialist

VMware SVGA II

- VMware virtual GPU takes the form of an emulated PCI device
  - VMware SVGA II
  - No physical instance of the card exists
- A device driver is provided for common Guests
  - Windows ones support 3D acceleration
- A user-level device emulation process is responsible for handling accesses to the PCI configuration and I/O space of the SVGA device

SVGA Device Architecture

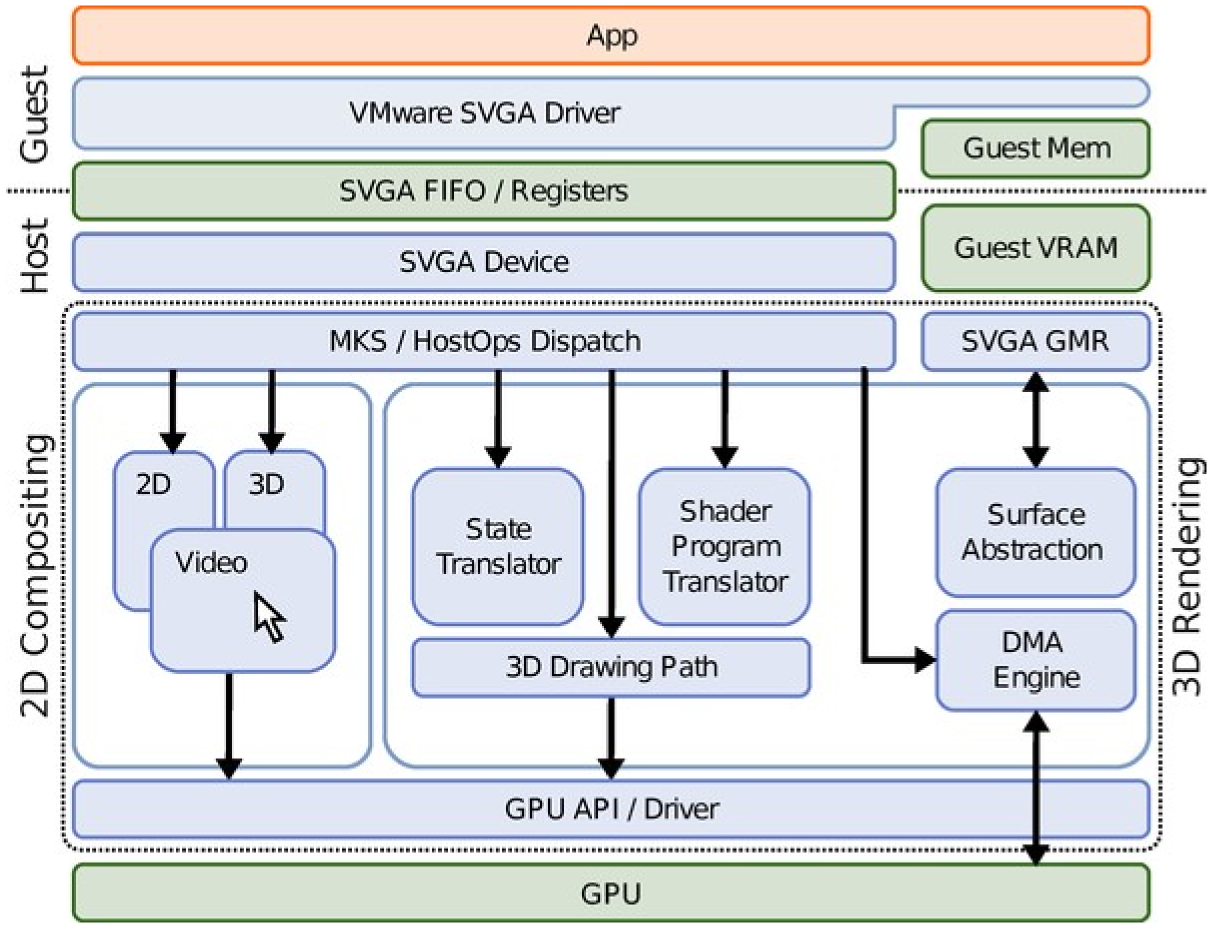
KNOWING YOU'RE SECURE



http://www.usenix.org/event/wiov08/tech/full_papers/dowty/dowty.pdf

The Virtual Graphic Stacks

KNOWING YOU'RE SECURE



http://www.usenix.org/event/wiov08/tech/full_papers/dowty/dowty.pdf

Memory-mapped I/O
(from Wikipedia)

KNOWING YOU'RE SECURE

- *Memory-mapped I/O* (MMIO) and *port I/O* (also called port-mapped I/O or PMIO) are two complementary methods of performing input/output between the CPU and peripheral devices in a computer
  - Each I/O device monitors the CPU's address bus and responds to any CPU's access of device-assigned address space
  - Port-mapped I/O uses a special class of CPU instructions specifically for performing I/O

My Simplified Version

Host

vmware-vmx Process

- SVGA FIFO
- Frame Buffer

Guest

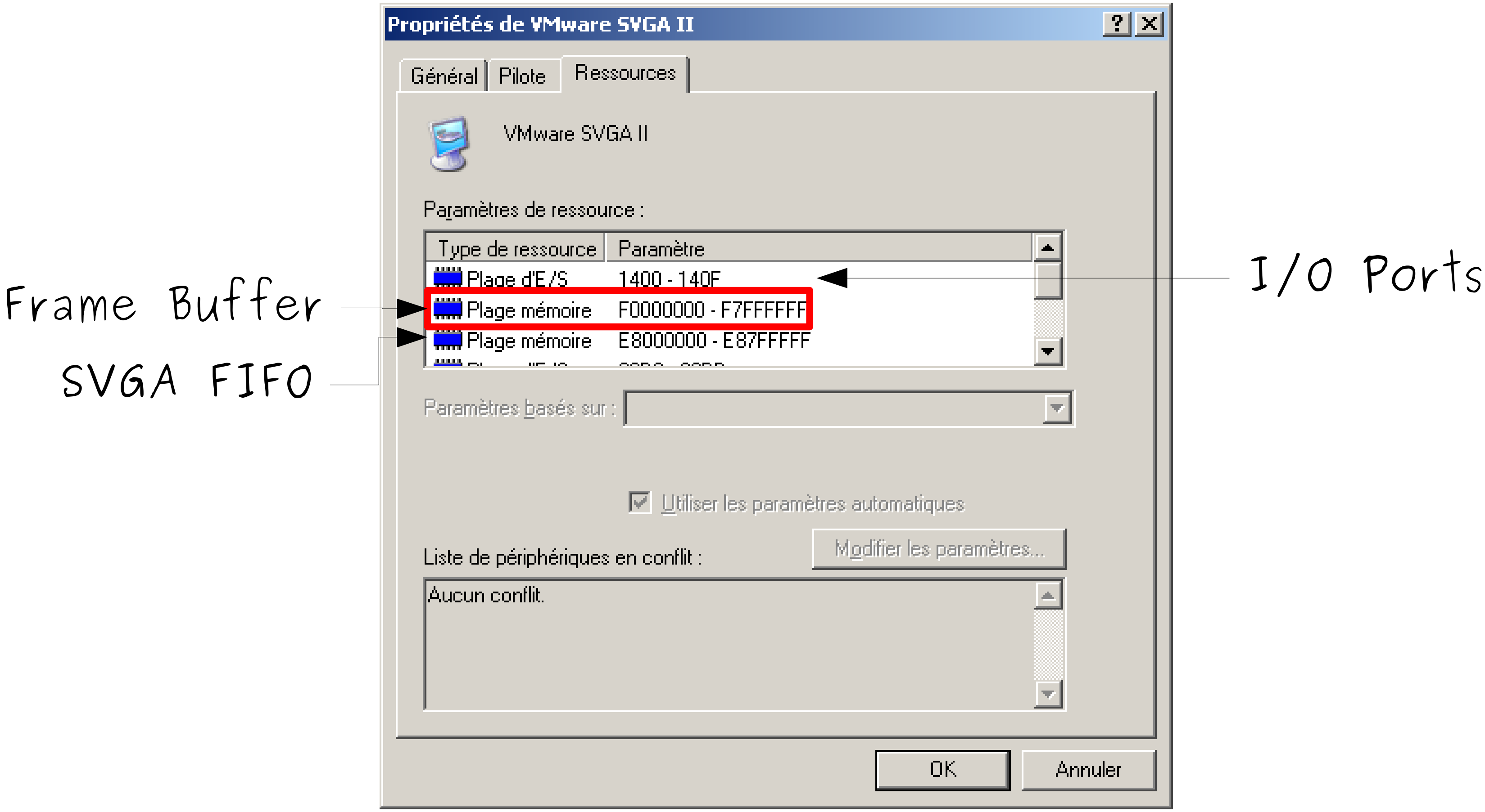
Virtual Machine

- I/O Ports
- I/O Memory Mappings

Virtual Video Card

VMware SVGA I/O

KNOWING YOU'RE SECURE



Windows 2003 SP1 (WKS)

SVGA FIFO

SVGA FIFO

- The SVGA device processes commands asynchronously via a lockless FIFO queue
  - This queue (several MB) occupies the bulk of the FIFO Memory region
- During unaccelerated **2D** rendering: FIFO commands are used to mark changed regions in the frame buffer
- During **3D** rendering: the FIFO acts as a transport layer for an architecture independent SVGA3D rendering protocol

2D FIFO Operations

KNOWING YOU'RE SECURE

- They can be found in xf86-video-vmware
- Sample 2D operations:
  - SVGA_CMD_UPDATE (1)
    - FIFO layout: X, Y, Width, Height
  - SVGA_CMD_RECT_FILL (2)
    - FIFO layout: Color, X, Y, Width, Height
  - SVGA_CMD_RECT_COPY (3)
    - FIFO layout: Source X, Source Y, Dest X, Dest Y, Width, Height
  - ...

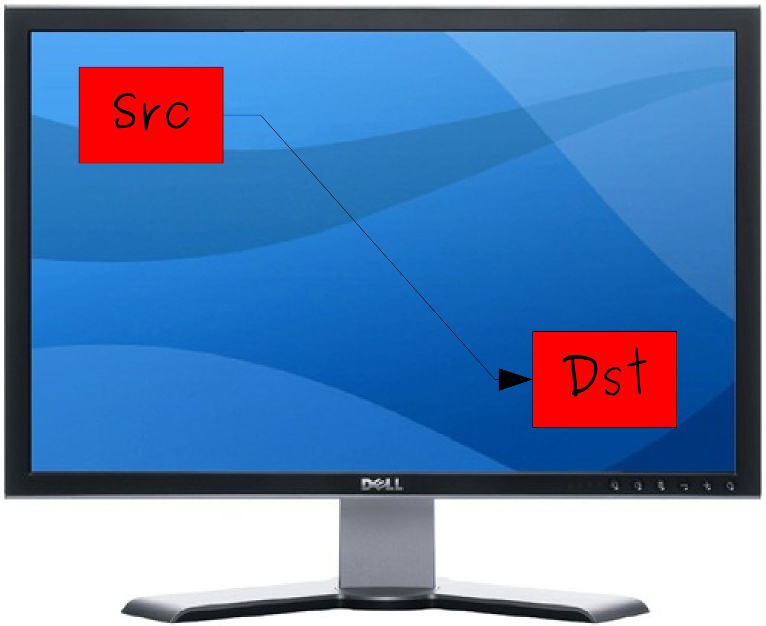
SVGA FIFO 2D Operations

KNOWING YOU'RE SECURE

SVGA_CMD_INVALID_CMD
SVGA_CMD_UPDATE
SVGA_CMD_RECT_FILL
SVGA_CMD_RECT_COPY
~~SVGA_CMD_DEFINE_BITMAP~~
~~SVGA_CMD_DEFINE_BITMAP_SCANLINE~~
~~SVGA_CMD_DEFINE_PIXMAP~~
~~SVGA_CMD_DEFINE_PIXMAP_SCANLINE~~
~~SVGA_CMD_RECT_BITMAP_FILL~~
~~SVGA_CMD_RECT_PIXMAP_FILL~~
~~SVGA_CMD_RECT_BITMAP_COPY~~
~~SVGA_CMD_RECT_PIXMAP_COPY~~
~~SVGA_CMD_FREE_OBJECT~~
SVGA_CMD_RECT_ROP_FILL
SVGA_CMD_RECT_ROP_COPY
~~SVGA_CMD_RECT_ROP_BITMAP_FILL~~
~~SVGA_CMD_RECT_ROP_PIXMAP_FILL~~
~~SVGA_CMD_RECT_ROP_BITMAP_COPY~~
~~SVGA_CMD_RECT_ROP_PIXMAP_COPY~~
SVGA_CMD_DEFINE_CURSOR
~~SVGA_CMD_DISPLAY_CURSOR~~
~~SVGA_CMD_MOVE_CURSOR~~
SVGA_CMD_DEFINE_ALPHA_CURSOR
SVGA_CMD_DRAW_GLYPH
SVGA_CMD_DRAW_GLYPH_CLIPPED
SVGA_CMD_UPDATE_VERBOSE
SVGA_CMD_SURFACE_FILL
SVGA_CMD_SURFACE_COPY
SVGA_CMD_SURFACE_ALPHA_BLEND
SVGA_CMD_FRONT_ROP_FILL
SVGA_CMD_FENCE
SVGA_CMD_VIDEO_PLAY_OBSOLETE
SVGA_CMD_VIDEO_END_OBSOLETE
SVGA_CMD_ESCAPE

SVGA_CMD_RECT_COPY

- Copies a rectangle in the Frame Buffer from a source X, Y to a destination X, Y

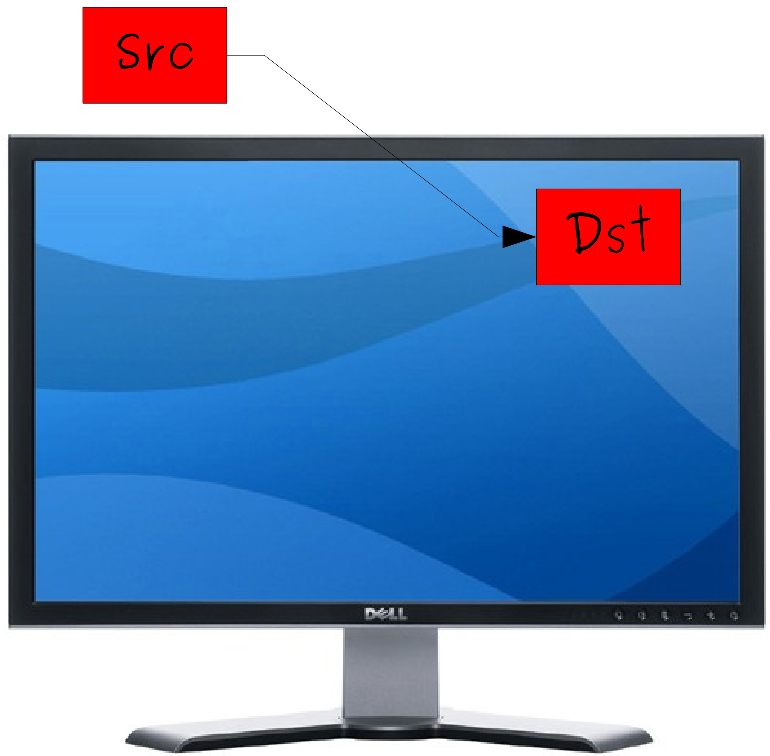


Frame Buffer

SVGA_CMD_RECT_COPY

KNOWING YOU'RE SECURE

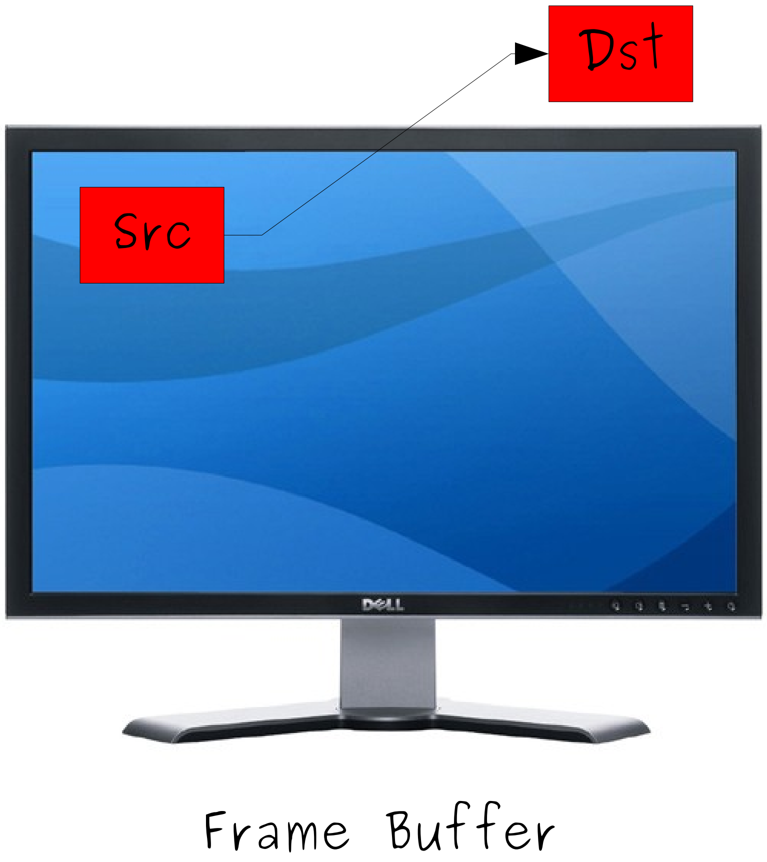
- Boundaries checks on the source location can be bypassed



Frame Buffer

SVGA_CMD_RECT_COPY

- Boundaries checks on the destination location can be bypassed (to a lower extent than source)



SVGA Arbitrary Read&Write

KNOWING YOU'RE SECURE

- Guest can *read* and *write* in the frame buffer
- Frame buffer is mapped in the host memory
- SVGA_CMD_RECT_COPY bugs mean:
  - One can copy host process memory into the frame buffer and read it
    - **Default unlimited arbitrary read**
  - One can write data into the frame buffer and copy it into the host process memory
    - **Default limited arbitrary write**
      - Only into the page preceding the frame buffer
      - *Might* be exploitable in some cases
        - Depends on what is mapped before the frame buffer

SVGA_CMD_DRAW_GLYPH

KNOWING YOU'RE SECURE

- Draws a glyph into the frame buffer
- Requires `svga.yesGlyphs="TRUE"`

Virtual Screen

SVGA_CMD_DRAW_GLYPH

KNOWING YOU'RE SECURE

- There is no check on the X, Y where the glyph is to be copied

Virtual Screen

Arbitrary WriteN

KNOWING YOU'RE SECURE

- Frame buffer is mapped in the host memory
- SVGA_CMD_DRAW_GLYPH bug means:
  - One can write any data, anywhere in the host process memory
    - Write address is relative to the base of the frame buffer
      - Pretty steady in ESX
      - Can be leaked with SVGA_CMD_RECT_COPY bug
    - **Non-default arbitrary write**
      - Fully exploitable

VMware & 3D

KNOWING YOU'RE SECURE

- Experimental 3D support appeared in VMware Workstation 5.0 (April 2005)
  - Disabled by default
  - Option had to be added to the config file of the VM
- It became **default** with Wks 6.5 (and Fusion?)
  - "*Accelerate 3D Graphics*" checkbox under Display
    - Code is reachable regardless of checkbox
- 3D operations are default and parsed under ESX 4.0 RC Hardfreeze

SVGA3D

- The SVGA3D protocol is a simplified and idealized adaptation of the Direct3D API
- It has a minimal number of distinct commands
- It is not publicly documented (AFAIK)
  - xf86-video-vmware has definitions for some constants but no prototypes of functions
- It uses "contexts" like Direct3D
  - Stored on the Host
  - Hold render states, light data, etc.

SVGA FIFO 3D Operations

KNOWING YOU'RE SECURE

SVGA_CMD_SURFACE_DEFINE
SVGA_CMD_SURFACE_DESTROY
SVGA_CMD_SURFACE_COPY
SVGA_CMD_SURFACE_DOWNLOAD
SVGA_CMD_SURFACE_UPLOAD
SVGA_CMD_INDEX_BUFFER_DEFINE
SVGA_CMD_INDEX_BUFFER_DESTROY
SVGA_CMD_INDEX_BUFFER_UPLOAD
SVGA_CMD_VERTEX_BUFFER_DEFINE
SVGA_CMD_VERTEX_BUFFER_DESTROY
SVGA_CMD_VERTEX_BUFFER_UPLOAD
SVGA_CMD_CONTEXT_DEFINE
SVGA_CMD_CONTEXT_DESTROY
SVGA_CMD_SETTRANSFORM
SVGA_CMD_SETZRANGE
SVGA_CMD_SETRENDERSTATE
SVGA_CMD_SETRENDERTARGET

SVGA_CMD_SETTEXTURESTATE
SVGA_CMD_SETMATERIAL
SVGA_CMD_SETLIGHTDATA
SVGA_CMD_SETLIGHTENABLED
SVGA_CMD_SETVIEWPORT
SVGA_CMD_SETCLIPPLANE
SVGA_CMD_CLEAR
SVGA_CMD_PRESENT
SVGA_CMD_DRAWPRIMITIVES
SVGA_CMD_DRAWINDEXEDPRIMITIVES
SVGA_CMD_SHADER_DEFINE
SVGA_CMD_SHADER_DESTROY
SVGA_CMD_SET_VERTEXSHADER
SVGA_CMD_SET_PIXELSHADER
SVGA_CMD_SET_SHADER_CONST
SVGA_CMD_DRAWPRIMITIVES2
SVGA_CMD_DRAWINDEXEDPRIMITIVES2

3D Bugs

- Many SET commands are flawed
- SETRENDERSTATE
  - The code:

```
.text:0065EE25
.text:0065EE25 loc_65EE25: ; CODE XREF: SetRenderStateInContext+25j
.text:0065EE25 mov edi, [ecx+eax*8]   ; Offset @ InputData[i]
.text:0065EE28 mov ebx, [ecx+eax*8+4] ; Data @ InputData[i+1]
.text:0065EE2C add eax, 1             ; i++
.text:0065EE2F cmp eax, edx
.text:0065EE31 mov [esi+edi*4+50h], ebx
.text:0065EE35 jb short loc_65EE25
```

  - Write primitive relative to **`esi`**
    - It's the context address in the host memory
    - It can be leaked in the guest thanks to the COPY bug!

Relative to Absolute

KNOWING YOU'RE SECURE

- SETLIGHTENABLED
  - The code:

```
.text:0065EF33 mov ecx, [ebp+arg_4]
.text:0065EF36 mov eax, [ecx+4]
.text:0065EF39 mov ecx, [ecx+8]
.text:0065EF3C mov edx, eax
.text:0065EF3E shl edx, 4
.text:0065EF41 sub edx, eax
.text:0065EF43 mov eax, [ebp+arg_0]
.text:0065EF46 mov eax, [eax+648h]
.text:0065EF4C mov [eax+edx*8], ecx
```

  - By overwriting Context+648h with the relative write, we get an absolute write primitive
  - Also works with SETLIGHTDATA for 29*4 bytes

Moar Bugz

- Additional bugs in:
  - SETRENDERTARGET
    - Signed bounds checking
  - SETCLIPPLANE
    - No bounds checking
  - SETTRANSFORM
    - No bounds checking

Exploitation

Requirements

KNOWING YOU'RE SECURE

- We have to be able to read/write directly into the framebuffer and the FIFO
  - Direct3D has some APIs for that
    - Everything is checked and sanitized on the Guest side
  - The solution is to write our own driver
    - Sits on top of VMware video driver
      - It can be standalone though
      - Less coding to do this way
    - Maps the framebuffer and FIFO for direct, unrestricted access
- Requires Admin rights in the VM

Exploitation Process

KNOWING YOU'RE SECURE

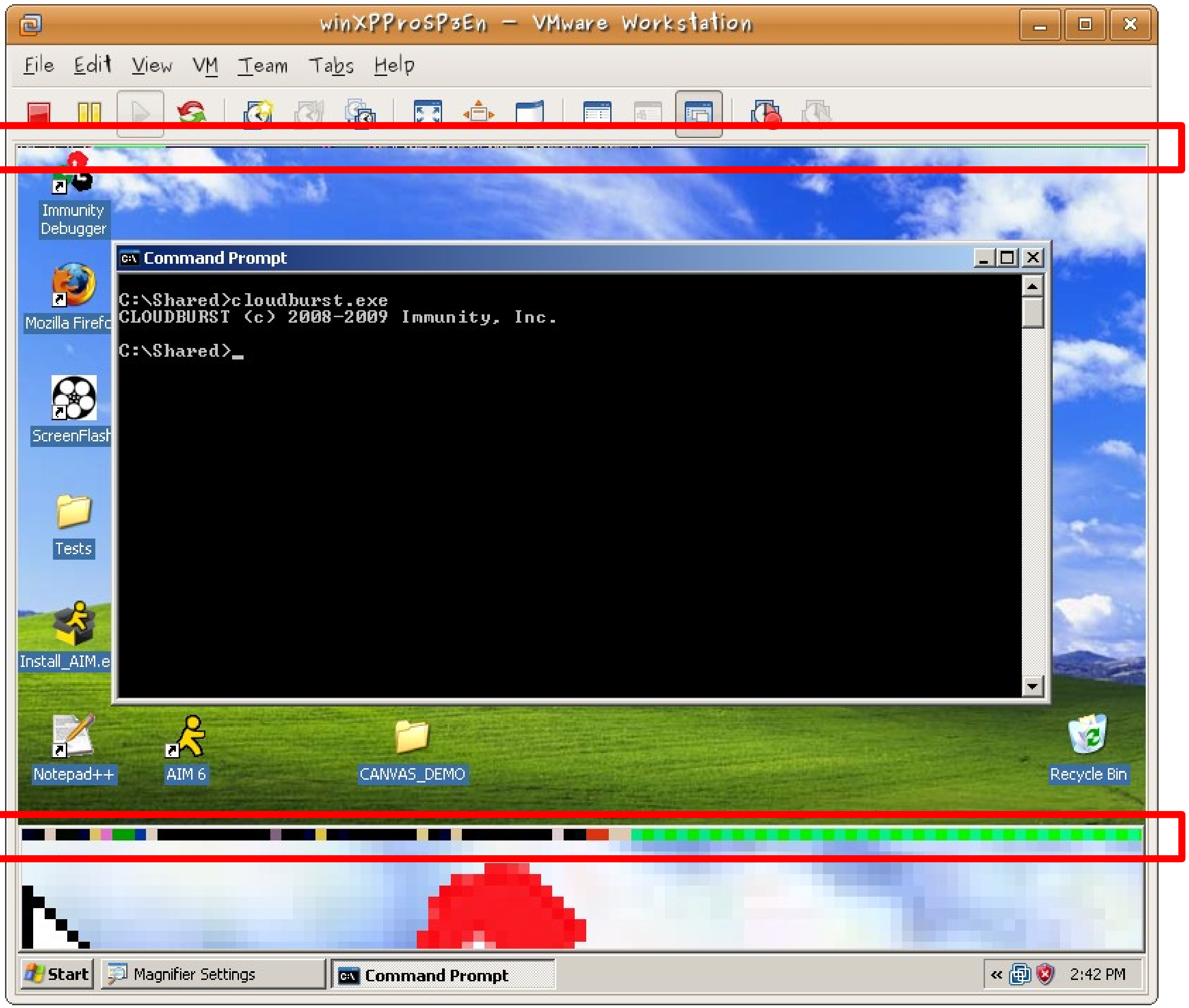
- Step #1: leak the base address of the framebuffer in the Host
  - All further leaks are relative to this address
- Some methods:
  - Windows Vista: relative memory leak
    - The page before the FB contains the address of the FB
  - Ubuntu: relative leak bruteforce
    - Keep leaking until your find the ELF header
  - Windows XP/Vista: absolute memory write
    - Then scan the FB for the data written
    - The FB is big enough to not trigger an access violation

Exploitation Process

KNOWING YOU'RE SECURE

- **Step #2**: fingerprint VMware version
  - We leak the PE/ELF header for that
    - They tend to be always at the same address
- **Step #3 to #n**: exploit ☺
  - Leak/Overwrite/Trigger/Leak/Overwrite/Trigger – Done!

Leak Example



We leak some data on the first line of the framebuffer (more visual)

Dealing with DEP

- When dealing with XP/Vista DEP AlwaysOn, or ESX 4.0 as a Host, we have to care about NX
- vmware-vmx provides VirtualProtect wrappers
  - One for RE, one for RW
  - They take their parameters in the .data section!
    - Easily abusable with the absolute write primitive
  - Also available for mprotect under Linux/ESX

Vista 12 Steps Example: 1 to 6

KNOWING YOU'RE SECURE

1) **Leak** the Frame Buffer Base address in the Host
2) **Leak** the PE Header of the vmware-vmx.exe binary
3) Based on the Timestamp in the PE Header, set the correct addresses needed
4) **Leak** the 1st pointer of the theSVGAUser structure
5) **Leak** the memory pointed by the leaked pointer to retrieve the address of the Context
6) **Overwrite** the VirtualProtect parameters so that the address is the one of the PE header and the size is 1000h. **Overwrite** as well the function pointer for the ESCAPE command with the address of the RW VirtualAlloc wrapper

Vista 12 Steps Example: 7 to 12

KNOWING YOU'RE SECURE

1) **Trigger** the ESCAPE command: the PE Header is now RW
2) **Write** the shellcode into the PE Header
3) Same as 6), except that we overwrite the ESCAPE function pointer with the RE VirtualAlloc wrapper
4) **Trigger** the ESCAPE command: the PE Header (and our shellcode) is now RE
5) **Overwrite** the ESCAPE function pointer with a pointer to our shellcode.
6) **Trigger** the ESCAPE command

KNOWING YOU'RE SECURE

MOSDEF Over Direct3D

(or how to tunnel a shell over BMP images)

MOSDEF

- MOSDEF (mose-def) is short for “Most Definately”
- MOSDEF is a retargetable, position independent code, C compiler that supports dynamic remote code linking written in pure Python
- In short, after you've overflowed a process you can compile programs to run inside that process and report back to you

Post Exploitation

KNOWING YOU'RE SECURE

- Ensure Host ⇔ Guest communication post exploitation, while not relying on extra features such as:
  - Network: Host can be unreachable from Guest
  - VMCI: not enabled by default
  - VMRPC: can be disabled
- Idea: tunnel the shell over the framebuffer
  - And in Ring3 to add some excitement

Guest Side: Direct3D API

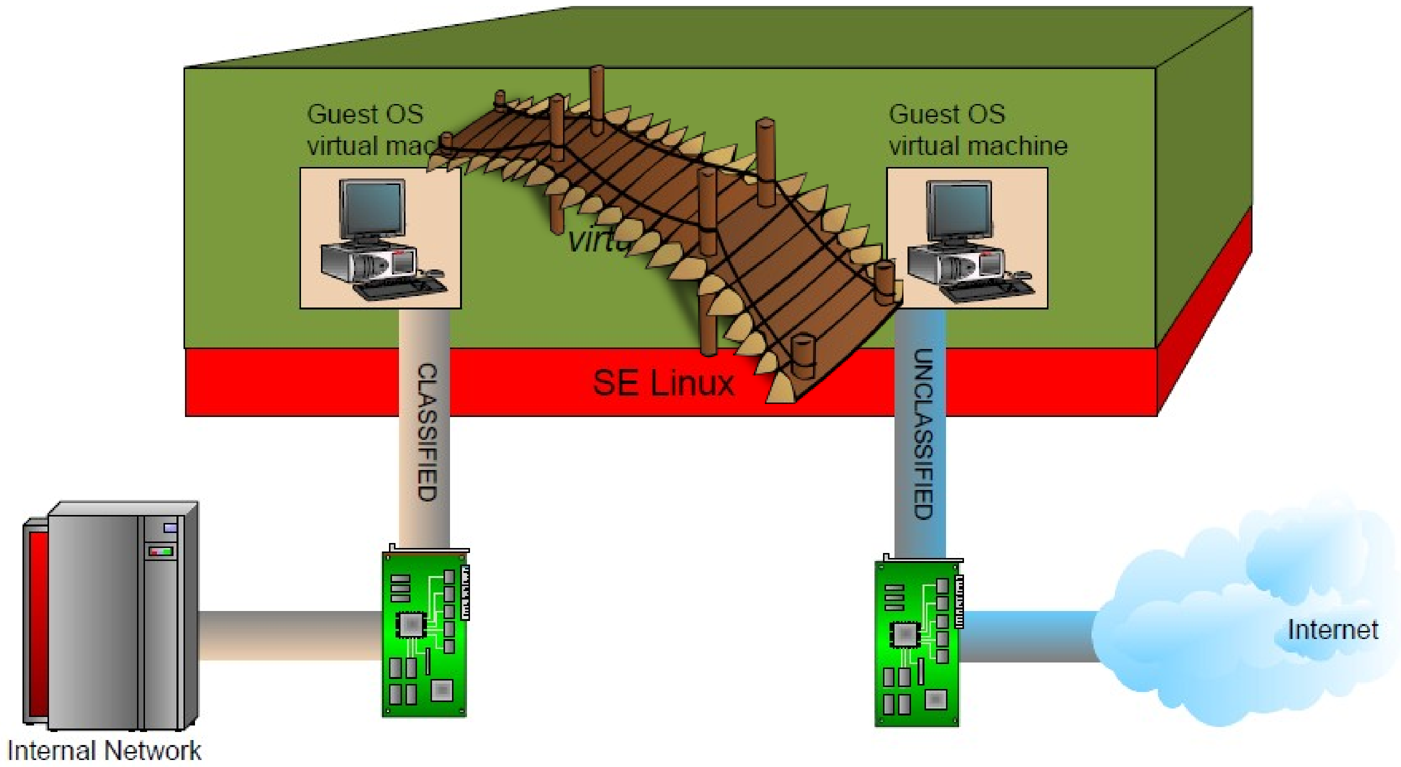
- Create and manipulate objects (surfaces) in the video card memory, off screen
  - CreateOffscreenPlainSurface
    - Format being D3DFMT_A8R8G8B8 (32 bits per pixel)
  - D3DXLoadSurfaceFromMemory
  - D3DXSaveSurfaceToFileInMemory
    - No “raw” format, use D3DXIFF_BMP
    - We parse the BMP to recover our data

Host Side

- Bind a MOSDEF listener on localhost
- Scan the video card memory for a "signature"
  - Extract and parse the data
  - Send it to the locally bound MOSDEF
  - Receive the result
  - Write it back to the framebuffer
- MOSDEF acting sequentially, we should not have any concurrent access issue
  - We implement a lousy "semaphore" to be sure

The Result

KNOWING YOU'RE SECURE



"Virtual Wooden Bridge"
over the
"Virtual Air Gap"

Conclusion

Who am I

- Title
  - Sr. Director VRT
- Industry Experience
  - 13+ Years
- Previous Companies
  - Farm9, Hiverworld (nCircle), IBM
- Certifications
  - I'll send you a PDF with all my credits, certs, and previous work. I'd open it in a VM.

☺

4

SOURCEfire
Security for the real world.

Virtualization Misconceptions

KNOWING YOU'RE SECURE

- VMware isn't an additional security layer
  - It's just another layer to find bugs in
- Given the correct bug primitives (memory leak, memory write), everything can be defeated
  - ASLR, NX
- Trying to patch silently in 2009 is ridiculous
- If a feature is not needed for a branch, the code shouldn't be included in it
  - Why would ESX ever need 3D support ...