CLOUDBURST

Last Updated June, 2nd 2009

IMMUNITY

Table of Contents

1 Introduction

This report presents the results of an auditing work carried out against VMware virtualization products in an attempt to find a way to execute code on the host from the guest.

The following is mostly focusing on the virtualized video device “VMware SVGA II” which happened to offer all the “features” needed to reliably execute code even on hosts with address space randomization and non-executable pages.

2 VMware SVGA II

The “VMware SVGA II” device is a virtualized PCI Display Adapter encountered in virtual machines run within any of the VMware products: VMware Workstation, VMware Server, VMware ESX and so on.

This device has a PCI Vendor ID of **0x15ad** and a PCI Product ID of **0x0405**. It replaced a while ago an older device “VMware SVGA” that had a PCI Product ID of **0x0710**.

This SVGA compatible controller is emulated on the host, and carries the graphical operations requested by the guest.

2.1 Memory Mapped I/O

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 (IN and OUT x86 instructions)

In the case of the VMware SVGA II device, we will mostly be dealing with **one** set of I/O ports (15 ports) and **two** MMIO address spaces. Under Windows, those can be found in the Advanced Properties of the Display Adapter, as shown in the Figure 1.

The later memory ranges are “shared” between the host and the guest. The Virtual Machine will write and read information there, while the graphic card emulated on the host will read and write in them to try and render the graphics.

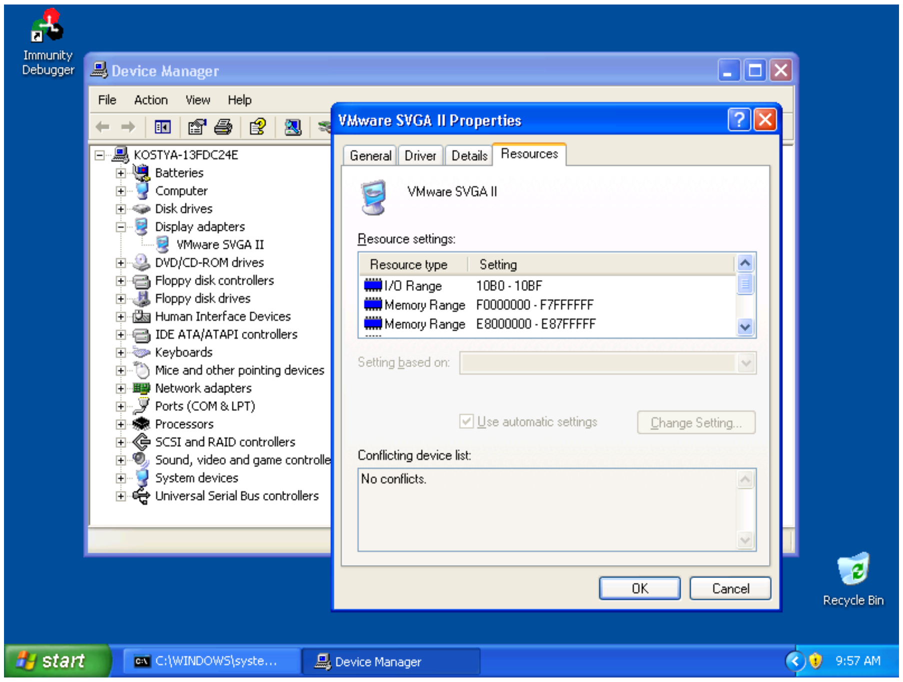
The two memory regions have specific purposes:

- The bigger one (although sizes can be configured) is the frame buffer. It is the memory region where the pixels are stored. Usually a pixel is stored on 4 bytes.

- The smaller one is the SVGA FIFO. The guest can store video commands in the FIFO that will be sequentially read and interpreted by the host.

This is basically represented in Figure 2.

The I/O ports are mostly used to read and write to 4 byte registers containing various information about the display.



VMware SVGA II

Resource settings:

Resource type	Setting
I/O Range	10B0 - 10BF
Memory Range	F0000000 - F7FFFFFF
Memory Range	E8000000 - E87FFFFF

Conflicting device list:

No conflicts.

Figure 1: I/O range and memory ranges for the VMware SVGA II controller

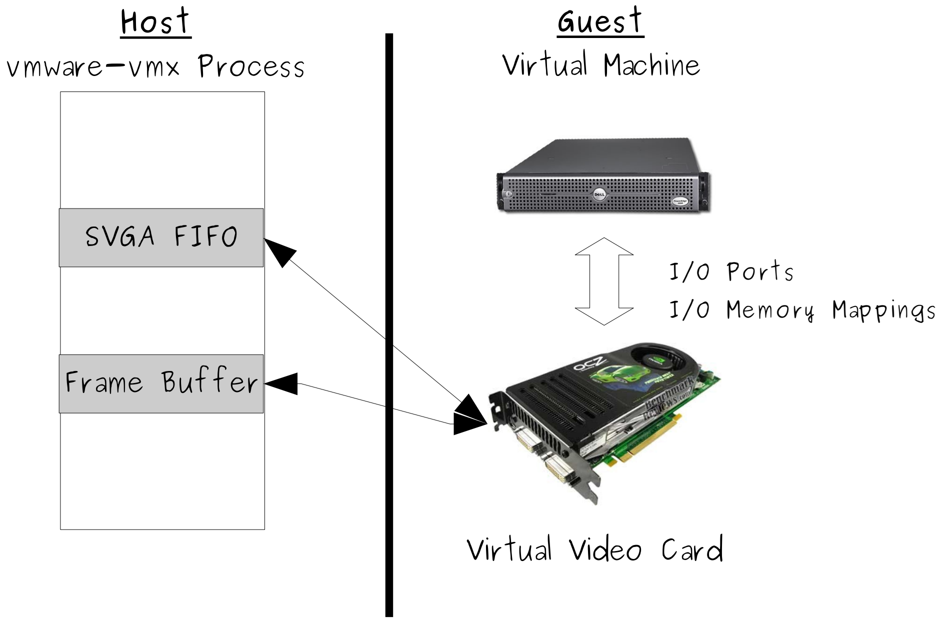


Figure 2: Representation of the shared memory regions

2.2 SVGA FIFO

Obviously nothing much can be done with the frame buffer since it will only be containing a bunch of pixels at any time. Yet the SVGA FIFO is a lot more interesting to have look at, since it can be written to by the guest and will be read from and parsed by the host. Parsing means potential bugs.

The way the SVGA FIFO works is undocumented. It is only supposed to be used by VMware video drivers, and the only few public references we could find to those are within the XF86 video drivers, written by VMware and open source.

The SVGA FIFO is mostly controlled by 4 4 byte integers (DWORD), located at the very beginning of the memory region that is the FIFO:

- SVGA_FIFO_MIN (0)
- SVGA_FIFO_MAX (1)
- SVGA_FIFO_NEXT_CMD (2)
- SVGA_FIFO_STOP (3)

Those are offsets indicated where the FIFO begins, ends, and where is the next command located. SVGA_FIFO_MIN and SVGA_FIFO_MAX tend not to be modified once the FIFO has be allocated and initialized in the host, but the guest can freely modify SVGA_FIFO_NEXT_CMD and SVGA_FIFO_STOP to indicate that new commands should be parsed.

The next piece of code illustrates how to add a DWORD in the FIFO:

```
void
vmwareWriteWordToFIFO(VMWAREPtr pVMWARE, CARD32 value)
{
    CARD32* vmwareFIFO = pVMWARE->vmwareFIFO;

    /* Need to sync? */
    if ((vmwareFIFO[SVGA_FIFO_NEXT_CMD] + sizeof(CARD32) ==
vmwareFIFO[SVGA_FIFO_STOP])
     || (vmwareFIFO[SVGA_FIFO_NEXT_CMD] == vmwareFIFO[SVGA_FIFO_MAX] -
sizeof(CARD32) &&
         vmwareFIFO[SVGA_FIFO_STOP] == vmwareFIFO[SVGA_FIFO_MIN])) {
        VmwareLog(("Syncing because of full fifo\n"));
        vmwareWaitForFB(pVMWARE);
    }

    vmwareFIFO[vmwareFIFO[SVGA_FIFO_NEXT_CMD] / sizeof(CARD32)] = value;
    if(vmwareFIFO[SVGA_FIFO_NEXT_CMD] == vmwareFIFO[SVGA_FIFO_MAX] -
       sizeof(CARD32)) {
        vmwareFIFO[SVGA_FIFO_NEXT_CMD] = vmwareFIFO[SVGA_FIFO_MIN];
    } else {
        vmwareFIFO[SVGA_FIFO_NEXT_CMD] += sizeof(CARD32);
    }
}
```

When the guest wants a new video operation to be done, it will add to the FIFO the various DWORDs forming the command and let the host know that the FIFO needs to be parsed.

A command is usually constituted of a number that identifies this command, and some parameters. Those parameters can be of fixed length, or of variable length depending on the command prototype. If of variable length, they are usually constituted of a length followed by some more DWORDs.

2.3 SVGA Commands

VMware implements a lot of video commands. They can be subdivided in two groups: the **2D** Commands and the **3D** Commands.

2D Commands appear to have been there since the very beginning of VMware products, 3D Commands have been added as non-default features quite recently and are now fully accessible in the latest version of VMware products: VMware Workstation 6.5 and above, VMware ESX Server 4.0 and above, etc.

Unfortunately, all the commands cannot be accessed by default. If a capability is present, then the command can be executed. Capabilities can be checked from the guest through a port I/O. Some capabilities are set as default, others must be activated through options in the VMX configuration file of the virtual machine.

2.3.1 2D Commands

ID	Command String	Parameters Number	Capabilities	FIFO Capabilities
0	INVALID_CMD	0	0	0
1	UPDATE	4	0	0
2	RECT_FILL	5	1	0
3	RECT_COPY	6	2	0
4	DEFINE_BITMAP	3 (var)	0	0
5	DEFINE_BITMAP_SCANLINE	4 (var)	0	0
6	DEFINE_PIXMAP	4 (var)	0	0
7	DEFINE_PIXMAP_SCANLINE	5 (var)	0	0
8	RECT_BITMAP_FILL	7	0	0
9	RECT_PIXMAP_FILL	5	0	0

0ah	RECT_BITMAP_COPY	9	0	0
0bh	RECT_PIXMAP_COPY	7	0	0
0ch	FREE_OBJECT	1	0	0
0dh	RECT_ROP_FILL	6	1	0
0eh	RECT_ROP_COPY	7	2	0
0fh	RECT_ROP_BITMAP_FILL	8	0	0
10h	RECT_ROP_PIXMAP_FILL	6	0	0
11h	RECT_ROP_BITMAP_COPY	0ah	0	0
12h	RECT_ROP_PIXMAP_COPY	8	0	0
13h	DEFINE_CURSOR	7 (var)	20h	0
14h	DISPLAY_CURSOR	2	20h	0
15h	MOVE_CURSOR	2	20h	0
16h	DEFINE_ALPHA_CURSOR	5 (var)	200h	0
17h	DRAW_GLYPH	5 (var)	400h	0
18h	DRAW_GLYPH_CLIPPED	0ah (var)	400h	0
19h	UPDATE_VERBOSE	5	0	0
1ah	SURFACE_FILL	7	1000h	0
1bh	SURFACE_COPY	9	1000h	0
1ch	SURFACE_ALPHA_BLEND	0ch	3000h	0
1dh	FRONT_ROP_FILL	6	8000h	0
1eh	FENCE	1	8000h	0
1fh	VIDEO_PLAY_OBSOLETE	0dh	0	0
20h	VIDEO_END_OBSOLETE	1	0	0
21h	ESCAPE	2 (var)	8000h	20h

Commands in GREY are deprecated (empty function of deprecated error).

Commands in **ORANGE** are (usually) not default.

(var) indicates that one of the parameters is of variable length.

Most of the commands are rather self explanatory. RECT_COPY for example takes 6 parameters: a source X and Y, a destination X and Y, a width, a height, and copies the source to the destination.

FIFO capabilities can be enabled in a rogue way from the guest.

2.3.2 3D Commands

ID	Command String	Parameters Number
410h	SURFACE_DEFINE	9
411h	SURFACE_DESTROY	1
412h	SURFACE_COPY	6
413h	SURFACE_STRETCHBLT	13h
414h	SURFACE_DMA	7
415h	CONTEXT_DEFINE	1
416h	CONTEXT_DESTROY	1
417h	SETTRANSFORM	12h
418h	SETZRANGE	3
419h	SETRENDERSTATE	1
41ah	SETRENDERTARGET	5
41bh	SETTEXTURESTATE	1
41ch	SETMATERIAL	13h
41dh	SETLIGHTDATA	1fh
41eh	SETLIGHTENABLED	3
41fh	SETVIEWPORT	5
420h	SETCLIPPLANE	6
421h	CLEAR	5
422h	PRESENT	1
423h	SHADER_DEFINE	3
424h	SHADER_DESTROY	3
425h	SET_SHADER	3
426h	SET_SHADER_CONST	8
427h	DRAW_PRIMITIVES	3
428h	SETSCISSORRECT	5
429h	BEGIN_QUERY	2

42ah	END_QUERY	4
42bh	WAIT_FOR_QUERY	4
42ch	PRESENT_READBACK	0
42dh	SVGA_3D_CMD_MAX	1

For 3D Commands, the deal is a bit different. Parameters are all of variable length. There is no capability check, all the 3D Commands are there.

Any 3D Command that has an ID between 3e8h and 40fh is a Legacy 3D Command. Those were removed from the current trees of VMware products.

3 CLOUDBURSTs

Only one of the vulnerabilities is described in this document!

3.1 RECT_COPY

Described in the xf86-video-vmware vmware.h file, the prototype of this function is:

```
#define  SVGA_CMD_RECT_COPY                       3
         /* FIFO layout:
            Source X, Source Y, Dest X, Dest Y, Width, Height */
```

It is available by default on all the tested VMware versions.

As expected, it will copy a source rectangle in the frame buffer to a given destination. A bunch of checks are carried out to ensure that some basic boundary conditions are respected. For example if Dest X + Width falls out the screen, the operation is said to "clip" and is aborted.

Yet the comparisons done on the DWORD and the results of the additions are **signed**. This opens the door to some malicious usage of the command. A quick graphic representation of what can happen is given in Figures 3 to Figure 5.

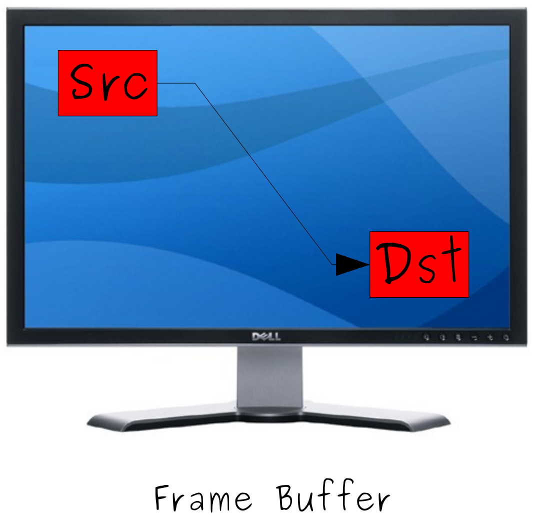


Figure 3: Normal behavior of the SVGA_RECT_COPY operation

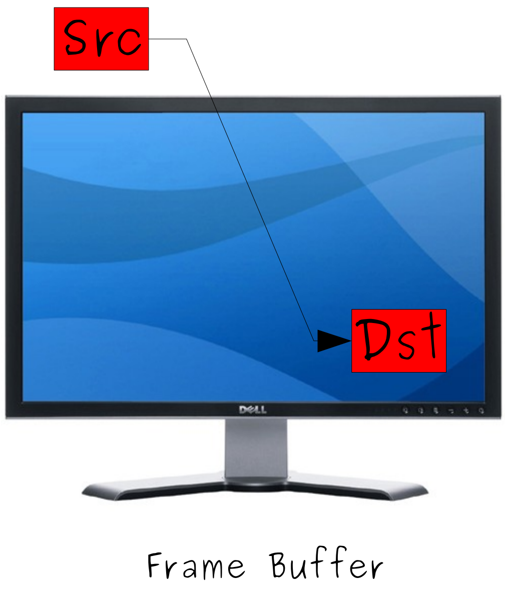


Figure 4: Source rectangle is out of the frame buffer (leak memory)

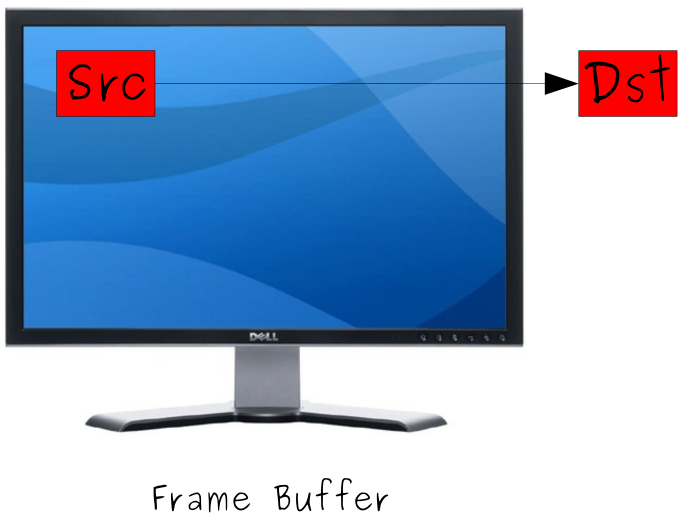


Figure 5: Destination rectangle is out of the frame buffer (overwrite memory)

There are two obvious ways to abuse the command, either misplace the source rectangle or the destination rectangle, leading to two different types of bugs.

3.1.1 Memory Leak (Figure 4)

If the source rectangle is located out of the frame buffer, the RECT_COPY operation will copy the content of the memory range **in the host process memory** defined as the source into the frame buffer. Since the frame buffer is shared between the host and the guest, the guest can then read the content of the frame buffer and thus leak the host process memory.

Debug versions (and Beta/RC) of VMware products include additional ASSERTs lowering the extent of the memory one can leak, even though the bug is still there. In retail versions, this bug can be used to leak pretty much any part of the memory.

The leak is relative to the base address of the frame buffer in the host process memory. In order to leak any address content, you will **HAVE** to know or leak this address.

3.1.2 Memory Write (Figure 5)

If the destination rectangle is abused in the RECT_COPY operation, it is possible for someone to overwrite part of the memory of the host process. Since the frame buffer is shared, a user can write some data into the frame buffer in the guest, then order the host to use this data as the source rectangle of the malicious copy. The result being a controlled overwrite, relatively to the base address of the frame buffer in the host process memory.

Much to our disappointment, it appears that the checks done on the destination are stricter than the ones being done on the source, which lowers the impact of the bug. We can **only** overwrite data in a few kilobytes of memory *preceding* our frame buffer.

Exploitability of this bug is highly dependent on the content of the memory pages right in front of the frame buffer in the host process memory. Practically it is most likely not exploitable under any platform.

Here is an idea on what to do:

```
VMwareWriteReg( &VMWARE, SVGA_REG_WIDTH, 0x7fffffff );
Width = VMwareReadReg( &VMWARE, SVGA_REG_WIDTH );
DbgPrint( "\nWidth:         0x%08x (%d)\n", Width, Width );
VMwareWriteReg( &VMWARE, SVGA_REG_CONFIG_DONE, 0 );
memset( VMWARE.FrameBuffer, 0x42, Width );
VMwareWriteWordToFIFO( &VMWARE, 0x03 );
VMwareWriteWordToFIFO( &VMWARE, 0x0 );
VMwareWriteWordToFIFO( &VMWARE, 0x0 );
VMwareWriteWordToFIFO( &VMWARE, 0x80000000 - Width );
VMwareWriteWordToFIFO( &VMWARE, 0x0 );
VMwareWriteWordToFIFO( &VMWARE, Width );
VMwareWriteWordToFIFO( &VMWARE, 0x1 ); // * BytesPerLine bytes
VMwareWriteReg( &VMWARE, SVGA_REG_CONFIG_DONE, 1 );
```

One can also play with the Y component to increase the offset where you overwrite the data. SVGA_REG_WIDTH is the limiting factor, and should be increased to get the most of this bug.

4 Post-Exploitation

Once code execution achieved on the Host, a communication problem still exists. We can't be sure that networking is enabled on the Host, or if it is that the Host is reachable from the Guest. We also cannot rely on features that might have been disabled by an administrator: VMRPC, VMCI (not enabled by default anyway), etc.

The shared memory regions between Host and Guest are a good starting ground to establish a canal between the two. And since we have been working all this time with the graphic features offered by VMware, we will continue in this way by using the Frame Buffer.

When trying to use the shared memory regions, we are facing some issues. They usually only are accessible through physical addressing, and thus in Ring0, which usually doesn't fit extremely well with Windows Sockets. And we probably also want something that doesn't require administrative privileges each time it is run.

While an I/O aware driver would also have worked, we decided to look in the direction of DirectX and Direct3D. The API set offered by Direct3D allows an unprivileged user to access the Frame Buffer off-screen areas through 3D surfaces and textures. The goal is then to proxy TCP/IP data (MOSDEF packets) from the Guest to the Host by using Direct3D APIs in the Guest (and simple memory read/right in the Host). The result can be defined as MOSDEF over Direct3D.

4.1 MOSDEF over Direct3D

4.1.1 Guest Side

On the Guest, we will be using Direct3D APIs to allocate some off-screen areas in the video card memory, write to them and read from them. In fact, only one of those areas is necessary to keep the communication canal working.

We need to work with the off-screen part of the video card memory and not the actual 2D Framebuffer (pixels on the screen) so that a user or screensaver won't corrupt our data.

The APIs we are using are:

- CreateOffscreenPlainSurface: creates an off-screen surface. This is a memory region in the video card memory, and thus shared between the Guest and the Host. As the format parameter, we use D3DFMT_A8R8G8B8 to use 32 bits per pixel.
- D3DXLoadSurfaceFromMemory: loads a surface from a memory buffer, ie: writes a buffer in the shared memory area from userland. Once again we use D3DFMT_A8R8G8B8 as format to make sure our entire buffer is copied (and not 3 bytes out of 4). D3DX_FILTER_NONE as the filter is also important to make sure that the data is not modified when being copied.
- D3DXSaveSurfaceToFileInMemory: closest counterpart we can get to the previous API. The surface will be saved from the video card memory to a buffer, unfortunately there is no such thing as a raw format, so we picked D3DXIFF_BMP, the easiest to parse and it doesn't involve compression.

All the APIs allow to specify a rectangle, so you can read or write only part of the surface, which will save some time and memory area later.

The process followed on the Guest is the following:

1. Create an off-screen surface
2. Connect-back to the MOSDEF listener (or Bind works too)
3. If there is some data to receive, receive it and load it into the surface
4. Else read the surface, if there is some Host data, send it to the listener

5. Loop to 3.

Since MOSDEF is designed to be sequential, there is no risk to have some data available in the surface when we are receiving some from the network socket.

4.1.2 Host Side

On the host side it is less complex since the frame buffer memory area is a regular memory region in regard of the vmware-vmx process.

A MOSDEF listener will be spawned and bind itself on localhost, waiting for data from the parent thread. The parent thread will scan the framebuffer memory area looking for new data from the Guest. Once such data has been found, it will be sent to the MOSDEF listener. It will then receive the reply and stored it into the frame buffer for the Guest to fetch.

4.1.3 Data Format

We chose to format the data in a simple way to allow Host and Guest to communicate easily. The data blocks are composed of a Signature (used to locate the Guest data in the framebuffer memory are from the Host side), a Flag (4 bytes), the length of the data (4 bytes) and the data.

The role of the Flag is the one of a semaphore, signaling data availability in the framebuffer. Flag can be either:

- NOOP (0), signaling that no data is available;
- GUEST_DATA (1), signaling to the Host that some data from the Guest is available to read;
- HOST_DATA(2), signaling to the Guest that some data from the Host is available to read;
- QUIT (3), signaling both ends should terminate the connection.

No data should be written to the framebuffer from either end if the Flag is not 0. The Flag should only be switched to 0 when all the data is read by the other end. The sequentiality of MOSDEF will ensure that no concurrent access is done on the communication canal.

5 Conclusion

With the results presented in this report, we were able to reliably execute code from the guest into the host on the following platforms:

- VMware Workstation 6.5.0 build-118166 on a Ubuntu Linux 8.04
- VMware Workstation 6.5.1 build-126130 on a Ubuntu Linux 8.04

- VMware Workstation 6.5.0 build-118166 on a Windows Vista SP1
- VMware Workstation 6.5.1 build-126130 on a Windows Vista SP1
- VMware ESX Server 4.0.0 build-133495

Workstation exploits obviously work flawlessly against VMware Players of the same build number.