!pool @eax Rants and ramblings on Windows internals and OS security

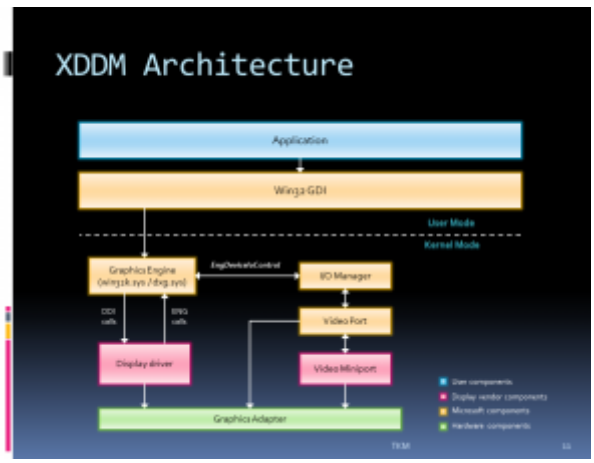
# Oracle VirtualBox Integer Overflow Vulnerabilities

Posted by kernelpool on July 19, 2011

In VirtualBox 4.0.10 and the Critical Patch Update for July 2011, Oracle addressed two vulnerabilities that could be leveraged by an attacker to gain elevated privileges in a Windows guest (CVE-2011-2300) or execute arbitrary code on the host (CVE-2011-2305). The former affected the VirtualBox XPDM display driver, installed on Windows guests as part of the VirtualBox Guest Additions, while the latter was a vulnerability in the VirtualBox 3D graphics stack, resulting in host memory corruption. Before we look closer at the details of these vulnerabilities, we'll briefly review the Windows 2000/XP display driver architecture.

### Windows 2000/XP Display Driver Model

In the XPDM architecture, every graphics adapter is associated with a display driver and a corresponding miniport driver. The display driver is essentially a DLL whose primary responsibility is rendering. It is written for any number of adapters that share a common drawing interface and hooks drawing operations offered by GDI (win32k) to enhance performance. The display driver also has direct access to video hardware registers for time-critical operations.




XP Display Driver Model Architecture

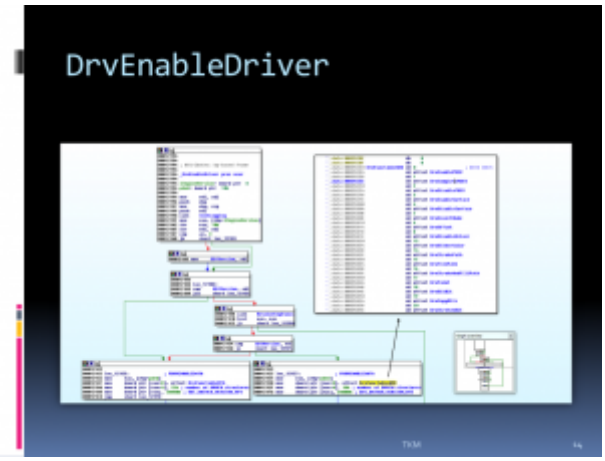
The video miniport driver generally interacts with other NT kernel components as is written specifically for one adapter (or family of adapters). It manages all the resources shared between the video miniport driver and the display driver, and handles hardware initialization, mode sets, and physical device memory mapping. Notably, the video miniport driver can only make calls exported by videoprt.sys. In order to request support from the video miniport driver, the display driver calls EngDeviceIoControl with a control code specific to the desired operation.

A display driver may implement a number of different graphics DDI functions depending on the drawing operations to accelerate or features to support. In general, DDI functions are grouped into three categories: functions required by every display driver, functions required under certain conditions, and

functions that are optional. The functions implemented and supported by a display driver are defined by DrvEnableDriver (DriverEntry), whose prototype is shown below.

```
1  BOOL DrvEnableDriver(
2      ULONG iEngineVersion,
3      ULONG cj,
4      __in  DRVENABLEDATA *pded
5  );
```

Upon loading the driver, the OS sets the *iEngineVersion* according to the version of GDI that is currently running (e.g. Windows 2000 or XP). The display driver then fills the DRVENABLEDATA structure accordingly with the supported interfaces. Subsequently, upon making GDI and DirectX calls, win32k.sys and dxg.sys call the corresponding function in the display driver.




DrvEnableDriver Example

Because the XPDM display driver model requires vendors to marshal great amounts of data in kernel-mode in handling large and complex Direct3D/DirectDraw structures, there is a significant probability of security vulnerabilities being present. It should be noted that the newer Windows Vista Display Driver Model (WDDM) tries to address this by moving such processing into a user-mode driver. In the event that GDI does not offer a suitable interface for say some specific graphics adapter capability, XPDM allows developers to use generic Escape calls. This enables user-mode applications to instruct the display driver to exercise specific functionality, much like the DeviceIoControl interface does in regular kernel-mode drivers. It is no secret that such interfaces should be carefully audited as they directly operate on user provided data and thus become a prime candidate for privilege escalation vulnerabilities.

### XPDM Display Driver Integer Overflow Vulnerability

The XPDM display driver in VirtualBox implements the DrvEscape interface to provide information about the mode of operation, but more importantly the ability to set the visible region of a guest's display. This is done by making a VBOXESC_SETVISIBLEREGION escape call (e.g. via ExtEscape) with a RGNDATA formatted buffer holding the array of rectangles that compose the region. The display driver copies this information to a buffer of its own before sending the information to the host.

[trunk/src/VBox/Additions/WINNT/Graphics/Video/disp/xpdm/VBoxDispDriver.cpp]

```
876  case VBOXESC_SETVISIBLEREGION:
877  {
878      LOGF(("VBOXESC_SETVISIBLEREGION"));
879      LPRGNDATA lpRgnData = (LPRGNDATA)pvIn;
880
```

```
881     if (cjIn >= sizeof(RGNDATAHEADER)
882         &&  pvIn
883         &&  lpRgnData->rdh.dwSize == sizeof(RGNDATAHEADER)
884         &&  lpRgnData->rdh.iType  == RDH_RECTANGLES
885         &&  cjIn == lpRgnData->rdh.nCount * sizeof(RECT) + sizeof(RGNDAT
886     {
887         DWORD   i;
888         PRTRECT pRTRect;
889         int     rc;
890         RECT   *pRect = (RECT *)&lpRgnData->Buffer;
891 
892         pRTRect = (PRTRECT) EngAllocMem(0, lpRgnData->rdh.nCount*sizeof(
893         if (!pRTRect)
894         {
895             WARN(("failed to allocate %d bytes", lpRgnData->rdh.nCount*s
896             break;
897         }
898 
899         for (i=0; i<lpRgnData->rdh.nCount; ++i)
900         {
901             LOG(("New visible rectangle (%d,%d) (%d,%d)",
902                  pRect[i].left, pRect[i].bottom, pRect[i].right, pRect[i
903             pRTRect[i].xLeft   = pRect[i].left;
904             pRTRect[i].yBottom = pRect[i].bottom;
905             pRTRect[i].xRight  = pRect[i].right;
906             pRTRect[i].yTop    = pRect[i].top;
907         }
```

The vulnerability addressed in VirtualBox 4.0.10 was an integer overflow in multiplying the number of rectangles (cRects) with the size of a RTRECT structure (equivalent to RECT), subsequently used to determine the size of the driver allocated buffer. This in turn resulted in a kernel pool overflow as the driver later used cRects in copying the rectangles. The latest maintenance release ensures that the multiplication does not wrap by casting sizeof(RECT) to an uint64_t and that the rectangle count does not exceed 1 million.

```
936 if (    cjIn >= sizeof(RGNDATAHEADER)
937     &&  pvIn
938     &&  lpRgnData->rdh.dwSize == sizeof(RGNDATAHEADER)
939     &&  lpRgnData->rdh.iType  == RDH_RECTANGLES
940     &&  (cRects = lpRgnData->rdh.nCount) <= _1M
941     &&  cjIn == cRects * (uint64_t)sizeof(RECT) + sizeof(RGNDATAHEADER))
942 {
```

Although there's a theoretical possibility of exploiting this vulnerability, the large copy makes it somewhat difficult. On MP systems, an attacker may be able to corrupt a kernel object or structure in the non-paged pool and trigger use of it before the copy reaches unmapped memory or triggers some other fault.

### VBoxSharedOpenGL Host Service Integer Overflow Vulnerability

As current graphics adapters do not offer virtualization of resources, providers of virtualization solutions need to abstract the graphics rendering in guests in order to offer the same 2D/3D experience you get on native systems. In VMware Workstation and Player, 3D rendering is performed by buffering commands in a shared memory between the guest and the host which the host processes and translates to OpenGL or Direct3D calls. VirtualBox takes a similar approach and uses Chromium to forward graphics calls from the guest system to the host where they are executed with hardware acceleration.

In the Host-Guest Communication Manager (HGCM), VirtualBox provides the VBoxSharedOpenGL service to give guest applications more direct control of commands passed to the host. Guest applications connect to the VBoxSharedOpenGL service by opening the VBoxGuest device and using the VBOXGUEST_IOCTL_HGCM_CONNECT I/O control code. Subsequent commands are then passed by issuing

HGCM calls (VBOXGUEST_IOCTL_HGCM_CALL) and specifying the service command to execute. One of the service commands (SHCRGL_GUEST_FN_WRITE_BUFFER) processed by the host service was found vulnerable to an integer overflow. The affected code is shown below.

[trunk/src/VBox/HostServices/SharedOpenGL/crserver/crservice.cpp]

```
793 case SHCRGL_GUEST_FN_WRITE_BUFFER:
794 {
795     Log(("svcCall: SHCRGL_GUEST_FN_WRITE_BUFFERn"));
796     /* Verify parameter count and types. */
797     if (cParms != SHCRGL_CPARMS_WRITE_BUFFER)
798     {
799         rc = VERR_INVALID_PARAMETER;
800     }
801     else
802     if (   paParms[0].type != VBOX_HGCM_SVC_PARM_32BIT /*iBufferID*/
803         || paParms[1].type != VBOX_HGCM_SVC_PARM_32BIT /*cbBufferSize*/
804         || paParms[2].type != VBOX_HGCM_SVC_PARM_32BIT /*ui32Offset*/
805         || paParms[3].type != VBOX_HGCM_SVC_PARM_PTR   /*pBuffer*/
806        )
807     {
808         rc = VERR_INVALID_PARAMETER;
809     }
810     else
811     {
812         /* Fetch parameters. */
813         uint32_t iBuffer      = paParms[0].u.uint32;
814         uint32_t cbBufferSize = paParms[1].u.uint32;
815         uint32_t ui32Offset   = paParms[2].u.uint32;
816         uint8_t *pBuffer      = (uint8_t *)paParms[3].u.pointer.addr;
817         uint32_t cbBuffer     = paParms[3].u.pointer.size;
818
819         /* Execute the function. */
820         CRVBOXSVCBUFFER_t *pSvcBuffer = svcGetBuffer(iBuffer, cbBufferSi
821         if (!pSvcBuffer || ui32Offset+cbBuffer>cbBufferSize)
822         {
823             rc = VERR_INVALID_PARAMETER;
824         }
825         else
826         {
827             memcpy((void*)((uintptr_t)pSvcBuffer->pData+ui32Offset), pBu
828
829             /* Return the buffer id */
830             paParms[0].u.uint32 = pSvcBuffer->uiId;
831         }
832     }
833
834     break;
835 }
```

In the above code, the VBoxSharedOpenGL service does not sufficiently validate the ui32Offset parameter used in specifying the offset into a host managed buffer. If ui32Offset+cbBuffer results in a wrapped integer, the function may attempt to copy the contents of the guest supplied data at a negative offset from the allocated buffer on x86 hosts (and thus corrupting the preceding memory), or at a positive unvalidated buffer offset (+ui32Offset) on x64 systems. In order to fix this, ui32Offset was cast to uint64_t.

Naturally, exploitability in this case depends on the memory allocator of the host operating system. Writing at a negative offset allows the attacker to corrupt the memory of any preceding memory allocation and may allow for arbitrary code execution if the attacker can sufficiently influence the heap state of the host process. On 64-bit hosts, the attacker may pass a very large pBuffer (cbBuffer) and thus require a small ui32Offset to pass the overflow check. This would allow the attacker to corrupt subsequently positioned memory blocks in the host process.

In attempting to exploit vulnerabilities in virtualized devices such as in a guest-to-host escape scenario, an attacker is typically required to install a driver (to do raw I/O operations or manipulate device memory directly) or perform other tasks that require administrative privileges. In this particular case, no special privileges are required as the vulnerability is triggered through the VirtualBox guest device which permits unprivileged access. Although Oracle admits that the Chromium stack (not enabled by default) may open VirtualBox to security vulnerabilities, the affected component is not directly related to Chromium.

Vulnerabilities

← Mitigating Null Pointer Exploitation on Windows

Windows Hooks of Death: Kernel Attacks through User-Mode Callbacks →

Comments are closed.

RSS Feed

Search: type, hit enter SEARCH

## Recent Posts

CVE-2012-0148: A Deep Dive Into AFD
Windows Hooks of Death: Kernel Attacks through User-Mode Callbacks
Oracle VirtualBox Integer Overflow Vulnerabilities
Mitigating Null Pointer Exploitation on Windows
Thread Desynchronization Issues in Windows Message Handling

## Recent Comments

## Archives

February 2012
August 2011
July 2011
February 2011
January 2011
December 2010

## Categories

Conferences
Exploitation
Mitigations
Papers
Vulnerabilities

## Meta

Log in
Entries RSS
Comments RSS
WordPress.org