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*)

Hypervisor

Management Layer

vmware-vmx

I/O

What’s going on here?

I/O

vmware tools libs

Guest

CPU

Guest

CPU

* very
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...

```c
#define BDOOR_CMD_APMFUNCTION
#define BDOOR_CMD_GETDISKGEO
#define BDOOR_CMD_GETPRTRLOCATION
#define BDOOR_CMD_SETPRTRLOCATION
#define BDOOR_CMD_GETSELFLENGTH
#define BDOOR_CMD_GETNEXTPIECE
#define BDOOR_CMD_SETSELLENGTH
#define BDOOR_CMD_SETNEXTPIECE
#define BDOOR_CMD_GETVERSION
#define BDOOR_CMD_GETDEVICELISTELEMENT
#define BDOOR_CMD_TOGGLEDEVICE
#define BDOOR_CMD_GETGUIOPTIONS
#define BDOOR_CMD_SETGUIOPTIONS
#define BDOOR_CMD_GETSCREENSIZE
#define BDOOR_CMD_MONITOR_CONTROL
#define BDOOR_CMD_GETHWVERSION

/* CPL0 only. */
```

Host <-> Guest Communication - Backdoor

• Invoking Backdoor functions is simple:

```
mov eax 564D5868h /* magic number */
mov ebx command-specific-parameter
mov cx command-number /* 1001e = RPC */
mov dx 5658h /* VMware I/O port */
in eax dx
```
Host <-> Guest Communication - Backdoor

Backdoor Channel

Low-bandwidth
RPCI
High-bandwidth

Other

TCLO

Backdoor Channel

Hypervisor (host)

Guest (vm)
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 `vmware-vmx.exe` and sprinkled throughout the open-vm-tools source
Host <-> Guest Communication - RPCI

```
lea r9, sub_140068360
lea r8, aTools_capab_17 ; "tools.capability.dnd_version"
lea rdx, aGuestDnDversion ; "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)

```c
#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:

```c
/* Basic request types */
typedef enum {
    MESSAGE_TYPE_OPEN,
    MESSAGE_TYPE_SENDSIZE,
    MESSAGE_TYPE_SENDPAYLOAD,
    MESSAGE_TYPE_RECVSIZE,
    MESSAGE_TYPE_RECVPAYLOAD,
    MESSAGE_TYPE_RECVSTATUS,
    MESSAGE_TYPE_CLOSE,
} MessageType;
```
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, 0x001e //MESSAGE_TYPE_CLOSE
mov edx, 0x5658
mov ebx, SIZE
in eax, dx
```
GuestRPC

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

```assembly
.rdata:0000000140773FA7
db 0
dq offset aGuestRpc ; "GuestRpc"
.dq offset GuestRPC_FUNCS
.align 20h
dq offset aDiskbackdoor ; "DiskBackdoor"
.dq offset DiskBackdoor_FUNCS
db 0
```

- If we look closely inside GuestRPC_FUNCS we will notice the following:

```assembly
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:

```python
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

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

```c
static PyMethodDef MyMethods[] =
{
  {"rpc_send", py_rpc_send, METH_VARARGS, NULL},
  {"rpc_send_unclose", py_rpc_send_unclose, METH_VARARGS, NULL},
  {NULL, NULL, 0, NULL}
};

PyMODINIT_FUNC initRPCSend(void)
{
  (void) Py_InitModule("RPCSend", MyMethods);
}

static PyObject* py_rpc_send(PyObject* self, PyObject* args)
{
  uint8_t *msg=NULL;
  int sz=0;
  if (!PyArg_ParseTuple(args, "z#", &msg,&sz)){
    printf("[X] FAILED! \n");
    return NULL;
  }
  rpc_send(msg,sz);
  Py_RETURN_NONE;
}
```
Tools Dev – Python, CTypes

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

```python
RPC_SEND_BUFFER = ctypes.create_string_buffer('\\x60', # pusha\
'\\xb8\\x68\\x58\\x4d\\x56', # mov  eax,0x564d5868
+--- 24 lines: '\xb9\x1e\x00\x00\x00' mov ecx,0x1e-------------------
)

_prototype = ctypes.CFUNCTYPE(DWORD, LPVOID, DWORD, use_last_error=True)

VirtualProtect(RPC_SEND_BUFFER, len(RPC_SEND_BUFFER), PAGE_EXECUTE_READWRITE, 0)

_rpc_send = _prototype(ctypes.addressof(RPC_SEND_BUFFER))

def rpc_send(buf):
    return _rpc_send(buf, len(buf))
```
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

- Framework
- Agent
- Mutator
- Host
- VMWare WorkStation

- Manage through vmrun
- Start
- Attach
- Monitor
- Send test cases
- vmx
Fuzzing the RPC Interface - InMemory

• Since we know exactly where 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:

```python
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:
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
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
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
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 ;)

![Diagram of DnD Packet](attachment:image.png)
VMware DnD UAF
Conclusion