VUPEN security

VUPEN Security – Private Exploits & PoCs Service

In-Depth Analysis of Microsoft Windows Paint JPEG Integer Overflow Vulnerability (MS10-005 / CVE-2010-0028)

Table of Contents

Introduction

A vulnerability exists in Microsoft Windows Paint when processing malformed JPEG images, which could be exploited to execute arbitrary code.

Tested Versions

The vulnerability was analyzed on Microsoft Windows XP SP3.

Fixed Versions

This vulnerability was fixed with the MS10-005 security update.

Technical Details

MS Paint is an image file editor which has been in Microsoft Products since 1985.

The JPEG format contains specifics blocs. One of them contains dimensions of the image. Those can be seen in the block which begins with the bytes 0xFF 0xCO.

For example in the following sample, dimensions are 1024x768:

```
FF D8 FF E0 [...]                /* beginning of the JPEG file */
FF C0 00 11 08 03 00 05 00 [...] /* 1280x768 (0x500 and 0x300) */
```

When loading a JPEG file with MS Paint, the function "GpJpegDecoder::Decode()" is called from the library GdiPlus.dll. This function performs several virtual calls to the main executable *"C:\WINDOWS\system32\mspaint.exe":*

```
.text:4EBA35AF ; public: virtual long __stdcall GpJpegDecoder::Decode(void)
.text:4EBA35AF ?Decode@GpJpegDecoder@@UAGJXZ proc near ; DATA XREF:
[...]
.text:4EBA35AF arg_0           = dword ptr  8
.text:4EBA35AF
.text:4EBA3665 push    edx
.text:4EBA3666 push    eax
.text:4EBA3667 call    dword ptr [ecx+18h] /* GetPixelDataBuffer */
```

This last function will then call "writeHeader()" from the same library:

```
.text:4ECE389C ; public: virtual long __stdcall
GpBmpEncoder::GetPixelDataBuffer(struct tagRECT const *, int, int, class BitmapData
*)
.text:4ECE389C ?GetPixelDataBuffer@GpBmpEncoder@@UAGJPBUtagRECT@@HHPAVBitmapData@@@Z
proc near
 [...]
.text:4ECE38AD call   ?WriteHeader@GpBmpEncoder@@AAGJPBUColorPalette@@@Z ; GpBmpEn-
coder::WriteHeader(ColorPalette const *)
```

WriteHeader will then call twice the function located at 0x1030f6d in *mspaint.exe*.

We can divide the parsing process into two main parts. The first one is an initialization which leads to the function above with fixed parameters supplied via "WriteHeader()". During this stage, an important structure will be built based on the picture's dimensions as we can see below:

```
; private: long __stdcall GpBmpEncoder::WriteHeader(struct ColorPalette const *)
?WriteHeader@GpBmpEncoder@@AAGJPBUColorPalette@@@Z proc near
[...]
.text:4ECE3689       movzx   eax, [ebp+var_1E]  /* indicator */
.text:4ECE368D       imul    eax, [ebx+24h]     /* Width of Image */
.text:4ECE3691       add     eax, 7
.text:4ECE3694       shr     eax, 3
.text:4ECE3697       add     eax, edx
.text:4ECE3699       and     eax, 0FFFFFFFCh
.text:4ECE369C       mov     [ebx+6Ch], eax     /* RESULT */
```

RESULT is a value computed from the image. Basically, according to our tests, RESULT is always equal to Width * 3.

The function handles 4 parameters. The first one is a pointer to a structure which will contain information to decode the JPEG file into BMP (such as the current offset pointer, size of buffer and so one). The third one is a size to be used during the copy session.

During the initialization, 0x0e bytes and 0x28 bytes will be copied:

```
; private: long __stdcall GpBmpEncoder::WriteHeader(struct ColorPalette const *)
?WriteHeader@GpBmpEncoder@@AAGJPBUColorPalette@@@Z proc near
[...]
.text:4ECE36B4        push    edx
[...]
.text:4ECE36BA        push    0Eh     /* 3rd parameter */
[...]
.text:4ECE36C5        push    edx
[...]
.text:4ECE36D1        push    eax
.text:4ECE36D2        call    dword ptr [ecx+10h] ; calling vulnerable function
[...]
.text:4ECE36EB        push    edx
.text:4ECE36EC        push    28h     /* 3rd parameter */
[...]
.text:4ECE36F1        push    edx
.text:4ECE36F2        push    eax
.text:4ECE36F3        call    dword ptr [ecx+10h] ; calling vulnerable function
```

Once this part is achieved, the execution reaches the second part, in which the image will be completely decoded into a BMP file. Thus, the vulnerable function will be called several times in a loop. The following figure shows passed parameters:

```
.text:4ECE394A        mov     eax, [esi+24h]  /* Height                          */
.text:4ECE394D        sub     eax, edi        /* EDI = current line (decrease) */
.text:4ECE394F        dec     eax
.text:4ECE3950        imul    eax, [esi+68h]  /* RESULT = Width * 3              */
.text:4ECE3954        add     eax, [esi+60h]  /* pointer = 0x36 [0xe + 0x28]    */
[...]
.text:4ECE393F        mov     esi, [ebp+arg_0]
[...]
.text:4ECE397C        mov     ecx, [esi+4]
[...]
.text:4ECE3981        push    ebx
.text:4ECE3982        push    dword ptr [esi+68h]
.text:4ECE3985        push    eax
.text:4ECE3986        push    ecx
.text:4ECE3987        call    dword ptr [edx+10h] ; calling vulnerable function
```

Then, we reach function at address 0x1030f6d in “mspaint.exe”:

```
.text:01030F6D ; public: virtual long __stdcall CBmpStream::Write(void const *, un-
signed long, unsigned long *)
.text:01030F6D ?Write@CBmpStream@@UAGJPBXKPAK@Z proc near ; DATA XREF:
.text:01007224o
 [...]
.text:01030F8C mov     ebx, [ebp+arg_0]      /* 1st parameter : structure */
.text:01030F8F mov     eax, [ebx+10h]        /* OFFSET */
.text:01030F92 mov     ecx, [ebx+0Ch]        /* SIZE BUFFER */
.text:01030F95 add     eax, esi              /* OFFSET + SIZE_COPY */
.text:01030F97 add     ecx, 0Eh
.text:01030F9A cmp     ecx, eax              /* Will reallocate if ECX < EAX */
.text:01030F9C jnb     short loc_1030FB1
.text:01030F9E add     eax, 0FFFFFFF2h
.text:01030FA1 push    eax             ; dwBytes
.text:01030FA2 mov     ecx, ebx
.text:01030FA4 call    ?ReAllocBuffer@CBmpStream@@QAEJK@Z;
CBmpStream::ReAllocBuffer(ulong)
.text:01030FA9 test    eax, eax
.text:01030FAB jnz     loc_1031057
```

The aim of this function is to fill a buffer with information of the decoded JPEG. At the end of multiple calls to the vulnerable function, a big buffer will be filled in.

To perform this operation, the function will try to compute the total amount of bytes for writing the complete BMP file. It will save the last OFFSET for the last line, and will perform an addition to this OFFSET with the remaining line to compute the total amount of the file.

```
.text:01030F6D ; public: virtual long __stdcall CBmpStream::Write(void const *, un-
signed long, unsigned long *)
.text:01030F6D ?Write@CBmpStream@@UAGJPBXKPAK@Z proc near ; DATA XREF:
[...]
.text:01031021 mov     edx, [ebx+10h]      /* offset to right from      */
.text:01031024 mov     ecx, [ebp+arg_8]    /* amount to copy now        */
.text:01031027 mov     esi, [ebp+arg_4]
.text:0103102A lea     edi, [edx+eax-0Eh]  /* Destination buffer (from EAX) */
.text:0103102E mov     eax, ecx
.text:01031030 shr     ecx, 2              /* DWORD copy */
.text:01031033 rep movsd                   /* copy to EAX + EDX */
```

Lines are written from the last to the first. The current offset in EDX represents the bottom of the buffer. The amount of the copy is always the same (Width * 3). It is then possible to trigger a buffer overflow by forging a huge offset called OFFSET and a size of copy called SIZE_COPY, such as OFFSET + SIZE_COPY will result in a small size.

Exploitation

When supplying specially crafted values, MS Paint will allocate a small buffer and will try to write SIZE_COPY bytes at the huge OFFSET value from the current buffer.

```
.text:01030F6D ; public: virtual long __stdcall CBmpStream::Write(void const *, un-
signed long, unsigned long *)
.text:01030F6D ?Write@CBmpStream@@UAGJPBXKPAK@Z proc near ; DATA XREF:
.text:01007224o
 [...]
.text:01030F8C mov     ebx, [ebp+arg_0]
.text:01030F8F mov     eax, [ebx+10h]        /* Huge value (nearly 0xFFFFFFFF) */
.text:01030F92 mov     ecx, [ebx+0Ch]        /* size of buffer */
.text:01030F95 add     eax, esi              /* Overflow triggered */
.text:01030F97 add     ecx, 0Eh
.text:01030F9A cmp     ecx, eax              /* This time no buffer is reallocated
.text:01030F9C jnb     short loc_1030FB1     /* because EAX has overflowed and
.text:01030F9E add     eax, 0FFFFFFF2h       /* then EAX < ECX
.text:01030FA1 push    eax
```

```
.text:01030FA2 mov     ecx, ebx
.text:01030FA4 call    ?ReAllocBuffer@CBmpStream@@QAEJK@Z;
CBmpStream::ReAllocBuffer(ulong)
.text:01030FA9 test    eax, eax
.text:01030FAB jnz     loc_1031057
```

When the copy is achieved, the destination buffer will be somewhere in the heap, which will produce a memory corruption.

```
.text:01031021 mov     edx, [ebx+10h]      /* Huge value */
.text:01031024 mov     ecx, [ebp+arg_8]    /* 3rd parameter */
.text:01031027 mov     esi, [ebp+arg_4]
.text:0103102A lea     edi, [edx+eax-0Eh]  /* Destination buffer somewhere */
.text:0103102E mov     eax, ecx
.text:01031030 shr     ecx, 2              /* DWORD copy */
.text:01031033 rep movsd
.text:01031035 mov     ecx, eax
.text:01031037 and     ecx, 3
.text:0103103A rep movsb
```

However, due to the different calculations, it not possible to reliable control the code flow and execute arbitrary code.

The provided proof-of-concept files have been tested with Windows XP SP3.

Detection

Before calling the vulnerable function "*CbmpStream::Write()" function* in "*mspaint.exe"* for the third time, the following value is computed to calculate the total size of the buffer that will contain the decoded file:

VALUE = [(Height – 1) * 3 * Width + 0x36 + 3 * Width] & 0xFFFFFFFF + 3 * Width

The 0x36 comes from the initialization stage (see technical details).

The first term of the above addition represents the total size of the buffer. As BMP files contain a header with 3 bytes per pixel (each for a color). The program will then take this size without a line (Height -1) and that would be the offset to write from (the last line). It will then add the remaining line to figure out the total amount of the buffer for allocation. After the overflow, this size will be very small. It will then copy the last line at the computed offset, which will corrupt the heap.

A JPEG image file can be considered malicious if VALUE overflows and is greater than 0x100000000.

References

VUPEN/ADV-2010-0338:
http://www.vupen.com/english/advisories/2010/0338

MS10-005:
http://www.microsoft.com/technet/security/bulletin/ms10-005.mspx

Changelog

2010-02-17: Initial release