FEATURE 1

METAFILE ART CLASS

Dennis Elser
Secure Computing Corporation, Germany

Just like its predecessor the Windows Metafile Format (WMF), the Enhanced Metafile Format (EMF) consists of descriptive commands for drawing an image rather than bitplanes of the rendered image itself. And just like malformed WMF files [1], Enhanced Metafiles have also proved to be susceptible to misappropriation.

This article provides a technical analysis of a recent remotely exploitable file format vulnerability within *Windows*' graphics device interface (GDI) [2].

PAINT BY NUMBERS

The idea behind WMF and EMF files is application and device independence, respectively: metafiles contain a sequence of records that guide a drawing device in how to render an image.

There is no serious difference in functionality between the two file formats other than EMF being truly device independent by maintaining its dimensions, shape and proportions [3].

All EMF records begin with a 32-bit field which is used to identify the record type and another 32-bit field for the size of the record; both fields are in Little-Endian byte order. Next, there is record-specific data of variable length. An EMF image always starts with a header record (type = 0x1), followed by a sequence of EMF data records and an end-of-file record (type = 0xE).

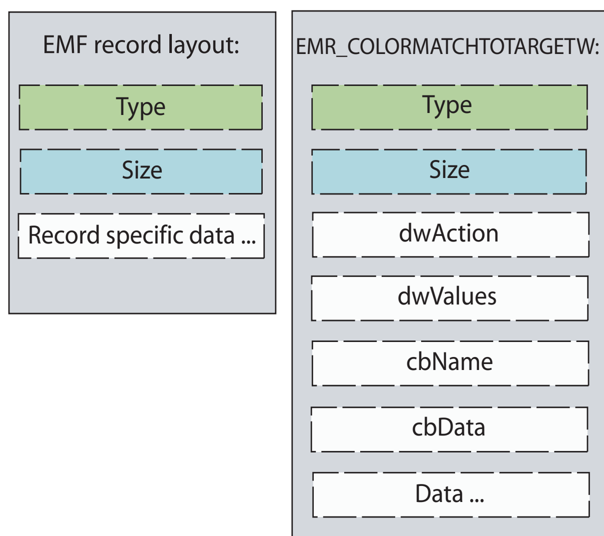

Figure 1: General structure of EMF records and the EMR_COLORMATCHTOTARGETW record.

A record's 'Size' field, as shown in Figure 1, is used by the parser as an offset to find the beginning of the next record (current record's file offset + 'Size'). However, there have also been EMF records with 'overlapping' data, as was the case with the first EMF exploit (MD5: 7DB16FD50CF76 CEF3d29DE47239C1F9A), which was found in the wild only two days after *Microsoft* published security bulletin MS08-021. Overlapping data wasn't relevant for the exploit's success, but was probably carelessness on the part of the exploit's author and a sign of intentionally corrupted data – easy to spot.

IMAGE INTERPRETATION

We can see three records in the exploit's hex-dump shown in Figure 2 below. Its green markers show the record's type; blue ones show the record's length. The exploit's first record type is EMR_HEADER (0x1), one of approximately 122 different record types. Its second is EMR_COLORMATCHTOTARGETW (0x79).

The exploit's third record (0xF1CF7512) is invalid, since the highest record-type number defined on current *NT*-based machines is 0x7A (wingdi.h). As we will see later, the reason for this is the second record's data exceeding its stated 'Size'.

```
0000 01000000 9C000000 8DFFFFFF B6FFFFFF
0010 EE0E0000 02170000 00000000 00000000
0020 FC510000 09740000 20454D46 00000100
0030 A4020000 2F010000 03000000 17000000
0040 6C000000 00000000 96120000 A11A0000
0050 C9000000 21010000 00000000 00000000
0060 00000000 DC120300 3D670400 00005600
0070 43006C00 61007300 73000000 50007200
0080 69006E00 74002000 70007200 65007600
0090 69006500 77000000 00000000 79000000
00A0 50000000 01000000 38000000 00000000
00B0 38000000 EB115B33 C966B985 014B8034
00C0 0B99E2FA 43FFE3E8 EAFFFFFF F9FD38A9
00D0 99999912 D9955E9F C1C1C1C4 5F9F5A12
00E0 E9853412 E9911875 999D9999 1275CFF1
00F0 17D79775 71639999 9910DC9D CFF1BC29
0100 665B7175 99999910 DC95CFF1 ...
```

Figure 2: Hex-dump of an EMF exploit.

By interpreting the structure of record type EMR_COLORMATCHTOTARGETW, which begins at file offset 0x9C in the exploit's hex-dump above, we can find out what causes the overflow. Each of the structure's fields has a size of 32 bits, with the 'Data' field being the only exception: it is an array of bytes that holds a Unicode name of a colour profile with additional raw colour profile data

```
mov     edi, ds:__imp__wcsncpy
push    ebx
mov     esi, 104h
push    esi                 ; size_t
lea     eax, [ebp+tempbuf]
push    offset _ColorDirectory ; wchar_t *
push    eax                 ; wchar_t *
call    edi ; __imp__wcsncpy
mov     ebx, ds:__imp__wcsncat
push    esi                 ; size_t
lea     eax, [ebp+tempbuf]
push    offset backslash ; "\\"
push    eax                 ; wchar_t *
call    ebx ; __imp__wcsncat
push    esi                 ; size_t
push    [ebp+Data]          ; wchar_t *
lea     eax, [ebp+tempbuf]
push    eax                 ; wchar_t *
call    ebx ; __imp__wcsncat
push    [ebp+destbuf_size] ; size_t
lea     eax, [ebp+tempbuf]
push    eax                 ; wchar_t *
push    [ebp+destbuf]       ; wchar_t *
call    edi ; __imp__wcsncpy
add     esp, 30h
pop     ebx
```

Figure 3: Broke implementation of 'BuildIcmProfilePath'.

appended. A colour profile is a file that contains information about the conversion of colours in context of a specific device [3].

In *Microsoft*'s specification, the size of a 'Data' field in bytes is appointed by the sum of the 'cbName' (0x0) and 'cbData' (0x38) fields, which in the exploit's case results in 0x38. So apparently, the exploit's 'Data' array at file offset 0xB4 is reserved 0 bytes for a colour profile name but 0x38 bytes for raw colour profile data. However, internal handling of this array of bytes looks a little different, as we will see below.

As soon as the *Windows* GDI parses an EMF file, its size fields are sanity checked (the sum of 'cbName' and 'cbData' must not int-overflow). Depending on the result of these checks, the particular record is flagged as either good or bad internally and only good ones are allowed further processing.

Once an EMR_COLORMATCHTOTARGETW record has passed this test, its 'Data' buffer is processed by 'BuildIcmProfilePath()', which is the function responsible for building a temporary colour profile's path on stack. However, the whole 'Data' buffer is interpreted as a Unicode string by the function without considering the size limitations provided by the 'Size' and 'cbName' fields. So even given a record length of 0x50, the

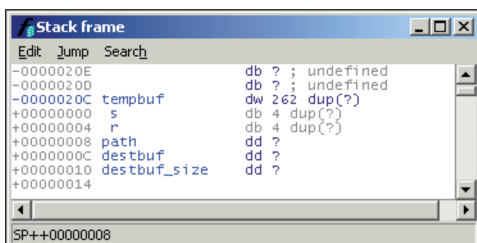

Figure 4: Stack frame of 'BuildIcmProfilePath'.

GDI's EMF parser keeps reading beyond the EMR_COLORMATCHTOTARGETW record's limits until a null-termination character is found within the 'Data' Unicode string.

ART FORGERY DETECTION

As both its partial disassembly (Figure 3) and stackframe (Figure 4) show, the maximum length of a path constructed by 'BuildIcmProfilePath()' was supposed to be restricted to 0x104 wide-characters (520 bytes). However, due to inappropriate use of wcsncat(), the actual maximum restriction is 3 * 0x104 wide characters in theory (1560 bytes), which is far more than can be stuffed into 'tempbuf'. 'Tempbuf' can hold a maximum of 262 wide characters including the null-terminating character.

This particular bug is found in early versions of gdi32.dll (i.e. *XP SP0*). Later versions of gdi32.dll (pre-MS08-021) succeed in calculating the remaining space of 'tempbuf' using wcslen(), but still fail to prevent the stack from being overwritten with 'Data' in a subsequent lstrcpyW() call found in 'IcmCreateColorSpaceByName()' – which reflects the vulnerability that has been fixed with MS08-021. Starting with that latest patch, the maximum number of wide characters copied to the stack is limited to 0x104.

In the end, with this limitation in effect, *Microsoft* deprecated its own EMF specification, since 'cbName' and 'cbData' are both 32-bit integers that allow far more bytes to be reserved for a colour profile.

It's not clear why a stack buffer of fixed size has been used instead of a heap buffer allowing for dynamic size, but the insights gained from the process of parsing the

```
IDA View-A
        mov     edi, edi
        push    ebp
        mov     ebp, esp
        sub     esp, 250h
        mov     eax, ___security_cookie
        mov     edx, [ebp+Data]
        push    esi
        mov     esi, [ebp+arg_0]
        mov     [ebp+var_4], eax
        push    edi
        xor     eax, eax
        mov     ecx, 93h
        lea     edi, [ebp+var_250]
        rep stosd
        and     [ebp+var_244], eax
        mov     eax, [ebp+arg_8]
        mov     [ebp+var_240], eax
        push    edx                         ; lpString2
        lea     eax, [ebp+var_20C]
        push    eax                         ; lpString1
        mov     [ebp+var_250], 50534F43h
        mov     [ebp+var_24C], 400h
        mov     [ebp+var_248], 24Ch
        call    ds:lstrcpyW(x,x)
0001AB61    77F2B761: IcmCreateColorSpaceByName(x,x,x,x)+3A
```

Figure 5: Unpatched EMF vulnerability.

EMF structure can now be used for defensive purposes. They allow us to build a generic detection mechanism for MS08-021-specific exploits by checking for EMR_COLORMATCHTOTARGETW records (and probably related ones) having a ‘Data’ field in relation to a filename with more than 0x104 wide characters.

SOFT-FOCUS EFFECT

With the record’s structure in mind and looking at the exploit’s hex-dump again (Figure 2), it is evident that ‘Data’ doesn’t contain a valid colour profile name and the number of wide characters in total exceeds 0x104 (the latter not being visible in the screenshot). Instead, the question of where exactly the exploit’s shellcode has been placed is answered by disassembling the record’s suspicious-looking ‘Data’ field at file offset 0xB4.

The shellcode (Figure 6) is XOR-encoded by an eight-bit key in order to avoid null-words, which could render the exploit ineffective when interpreted as zero-termination of Unicode strings. Once the shellcode has decoded itself on the stack, it uses the Process Environment Block (PEB) to find the base address of kernel32.dll in order to import several API functions by conducting a hashed string comparison of the APIs. These are used to download and install a backdoor, with a file name as ‘unambiguous’ as it can get – ‘word.gif’, from igloofamily.com (a domain hosted in Korea).

The shellcode possesses the ability to bypass modern behaviour blockers that detect API functions being called

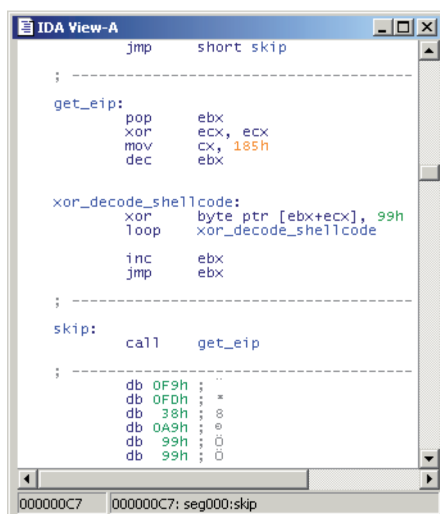

Figure 6: Shellcode found within an exploit’s EMR_COLORMATCHTOTARGETW record.

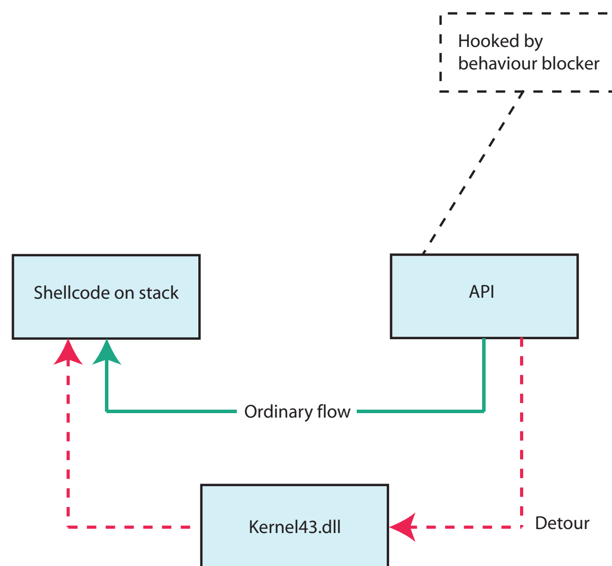

Figure 7: Behaviour blocker evasion mechanism.

from unusual places like the stack. This circumvention works by inserting a faked return address into the stack before calling an API function. The detoured return address points into a ‘ret’ assembler instruction within kernel32.dll, which in turn pops the real return address off the stack to be able to return control to the real caller. By doing so, a behaviour blocker is tricked into believing the actual caller is within the *Windows* kernel32 library area instead of shellcode on stack, thus being legitimate (Figure 7). The ‘ret’ instruction that is used as a detour is searched for manually within the kernel32.dll module near the address of WinExec().

In large part, this downloaded variant of the ‘Poisonivy’ backdoor consists of multi-layered, encrypted and position-independent code. Position-independent code not only makes it harder to read the disassembly, in this context it is necessary since the code is injected into the memory space of running processes such as ‘explorer.exe’ or ‘msnmsgr.exe’ (via WriteProcessMemory and CreateRemoteThread APIs).

As soon as the injected thread is executed, it creates a ‘hidden’ copy of the trojan as an NTFS alternate data stream (ADS) named ‘win_socks.exe’, attached to the ‘system32’ folder in the *Windows* directory. Its launch upon system reboot is ensured by the creation of an ‘Active Setup’ registry key in ‘HKCU\Software\Microsoft\Active Setup\Installed Components\{E5C1F9EA-A8FE-FCBC-9F3D-C2791859730F}’, named ‘StubPath’.

Afterwards the code stays in a loop waiting for *MSN Messenger* (msnmsgr.exe) to be run in order to inject another piece of self-decrypting code into it. This remote thread will initiate a connection with 'word.4pu.com', 'word1.4pu.com' and 'word2.4pu.com', giving the attacker total control over the compromised system. Since it is a program that is usually allowed to pass the firewall on most systems, *MSN Messenger* makes an ideal target for a backdoor like this variant of 'Poisonivy'.

CONCLUSION

Looking back at its history, the GDI has been a popular target among attackers. Besides the relatively recent EMF holes, GDI has also been vulnerable to remote attacks in the past, like the two ANImated cursor vulnerabilities (MS05-002 and MS07-017), a JPEG vulnerability in GDI+ (MS04-028) and WMF vulnerabilities (MS06-001). But thankfully there has been evolution: it's good to see the quality of the code going through different security stages, from using unsafe functions to using safer functions up to using proprietary safe functions in combination with the /GS compiler option.

As demonstrated by this article, it's not just lucrative business for the bad guys to perform binary code auditing and have a thorough look at the facts, but also for the good guys who need to defend these attacks. Vulnerabilities, like February's *Adobe Reader* vulnerability (CVE-2008-0655) and April's MS08-021 GDI vulnerability, begin to be exploited just a few days after the vendor's patch release – much faster than corporations may need to verify and deploy the patches. If we know the exact causes of vulnerabilities, we do not need to wait for the first exploits to fall into our hands in order to protect and defend ourselves effectively. Instead, we are able to tell apart benign structures from abnormal ones and thus can defend proactively against upcoming attacks.

REFERENCES

[1] Ferrie, P. Inside the Windows Meta File format. Virus Bulletin, February 2006, pp.5–8. http://www.virusbtn.com/vba/2006/02/vb200602-wmf.

[2] Microsoft Security Bulletin MS08-021. http://www.microsoft.com/technet/security/bulletin/ms08-021.mspx.

[3] Enhanced Metafile Format Specification Revision 2.0. http://msdn2.microsoft.com/en-us/library/cc204166.aspx.