2021/05/11_Windows32位内核_补课(进程内存数据的读、写-系统启动的过程-内核漏洞的利用-驱动的逆向)

笔记本:	Windows32位内核
创建时间:	2021/5/11 星期二 15:11
作者:	ileemi

课前会议

MmIsAddressValid(paddr); -- 检查地址是否有效（内部查询页表是否存在）

写内存时，设置中断，修改内存保护。读内存时，不需要修改内存保护。
页目录表在切换进程时会用到（KiSwapProcess）。
MmCreateHyperspaceMapping
MmGetPageTableForProcess

进程内存数据的读、写

CR3的获取和写入可通过 `__readcr3()`、`__writecr3()` 来实现。

ZwQuerySystemInformation
SYSTEM_INFORMATION_CALL
SYSTEM_PROCESS_INFORMATION
PsLookUpProcessByProcessId：通过PID返回对应的EPROCESS

```
NTSTATUS PsLookupProcessByProcessId(
        _In_ HANDLE ProcessId,
```

```
        _Outptr_ PEPROCESS *Process
        );
```

> PHANDLE_TABLE PspCidTable; -- 遍历这个链表可以将隐藏的进程遍历出来。

```
914: /*
915:  * @implemented
916:  */
917: NTSTATUS
918: NTAPI
919: PsLookupProcessByProcessId(IN HANDLE ProcessId,
920:                            OUT PEPROCESS *Process)
921: {
922:     PHANDLE_TABLE_ENTRY CidEntry;
923:     PEPROCESS FoundProcess;
924:     NTSTATUS Status = STATUS_INVALID_PARAMETER;
925:     PAGED_CODE();
926:     PSTRACE(PS_PROCESS_DEBUG, "ProcessId: %p\n", ProcessId);
927:     KeEnterCriticalRegion();
928:
929:     /* Get the CID Handle Entry */
930:     CidEntry = ExMapHandleToPointer(PspCidTable, ProcessId);
931:     if (CidEntry)
932:     {
933:         /* Get the Process */
934:         FoundProcess = CidEntry->Object;
935:
936:         /* Make sure it's really a process */
937:         if (FoundProcess->Pcb.Header.Type == ProcessObject)
938:         {
939:             /* Safe Reference and return it */
940:             if (ObReferenceObjectSafe(FoundProcess))
941:             {
942:                 *Process = FoundProcess;
943:                 Status = STATUS_SUCCESS;
944:             }
945:         }
```

可通过进程PID当作PspCidTable（全局）的数组下标，通过下标即可访问对应进程的 EPROCESS

MmGetDirectoryFrameFromProcess
MmGetPhysicalAddress：通过虚拟地址计算出物理地址
MmMapIoSpace：映射内存
ProbeForWrite：判断当前地址是否可以写入，不能写会抛异常。

cli // 屏蔽中断，不接收CPU信号，在eflag中有对应的标志，只能在Ring0中使用
sli // 恢复中断，接收CPU信号

代码示例：

```
#include <Ntifs.h>
#include <ntddk.h>

#ifdef DBG
  #define dprintf  DbgPrint
#else
  #define dprintf
#endif

/*
// 方法1：ULONG DirBase 页目录地址
// 方法2：PEPROCESS Process
UniqueProcess：进程ID
pAddress：读取的内存地址
pBuf：保存数据的缓冲区地址
```

```
uLength：读取的字节长度
*/
NTSTATUS MyReadProcessMemory(HANDLE UniqueProcess,
  void* pAddress,
  void* pBuf,
  ULONG uLength) {
  //ULONG OldCR3;
  NTSTATUS Status = STATUS_INVALID_ADDRESS;
  KAPC_STATE ApcState; // 会保存当前寄存器的状态
  PEPROCESS Process = NULL;

  dprintf("MyReadProcessMemory\n");

  __try {
    // 通过进程ID获取对应的EPROCESS
    Status = PsLookupProcessByProcessId(UniqueProcess, &Process);
    // 判断返回的EPROCESS是否有效
    if (NT_SUCCESS(Status)) {
      //
      // 检查缓冲区地址以及PID是否有效，物理地址不需要检测
      //
      if (MmIsAddressValid(pBuf) || MmIsAddressValid(Process)) {
        /* OldCR3 = __readcr3(); // 读取CR3并保存
         __writecr3(DirBase);

         // 判断进程中的内存是否有效
         if (MmIsAddressValid(pAddress)) {
           RtlCopyMemory(pBuf, pAddress, uLength);
           Status = STATUS_SUCCESS;
         }
         __writecr3(OldCR3);*/

        // 方法2（通过EPROCESS）And 方法3（通过PID获取EPROCESS）
        //切换进程，不在需要切换CR3
        KeStackAttachProcess(Process, &ApcState);

        // 判断进程中的内存是否有效
        if (MmIsAddressValid(pAddress)) {
          RtlCopyMemory(pBuf, pAddress, uLength);
          Status = STATUS_SUCCESS;
        }

        // 内存地址有效，不需要进行内存映射
        // 将目标地址的物理地址（需要指定进程）映射到当前进程中
        //MmMapIoSpace()

        //进程切换回来
```

```
        KeUnstackDetachProcess(&ApcState);
      }
    }
  }
  __except (1) {
    dprintf("MyReadProcessMemory __except (1)\n");
  }

  // 打开引用计数(ObOpenObjectByPointer)后需要 减1
  //ObDereferenceObject(Process);
  return Status;
}

/*
UniqueProcess: 进程ID
pAddress: 写入的内存地址
pBuf: 写入数据的缓冲区地址
uLength: 读取的字节长度
*/
NTSTATUS MyWriteProcessMemory(HANDLE UniqueProcess,
  void* pAddress,
  void* pBuf,
  ULONG uLength) {
  //ULONG OldCR3;
  NTSTATUS Status = STATUS_INVALID_ADDRESS;
  KAPC_STATE ApcState;
  PEPROCESS Process = NULL;
  PHYSICAL_ADDRESS pa = {0};  // 保存物理地址
  PVOID lpMapBuf = { 0 };     // 保存在内核中映射的地址

  dprintf("[51asm] %s %d %s\n", __FILE__, __LINE__, __FUNCTION__);

  __try {
    Status = PsLookupProcessByProcessId(UniqueProcess, &Process);
    if (NT_SUCCESS(Status)) {
      if (MmIsAddressValid(pBuf) || MmIsAddressValid(Process)) {
        /*
        // 方法1: 根据页目录表的地址读取内存
        OldCR3 = __readcr3();
         __writecr3(DirBase);

         if (MmIsAddressValid(pAddress)) {
           RtlCopyMemory(pBuf, pAddress, uLength);
           Status = STATUS_SUCCESS;
         }
         __writecr3(OldCR3);
         */
```

```
        //切换进程
        KeStackAttachProcess(Process, &ApcState);

        if (MmIsAddressValid(pAddress)) {
          // 判断当前地址是否可以写入
          //ProbeForWrite(pAddress, uLength, 4);
          /*
          // 方法1:
          // 修改内存保护属性,强制写入,寄存器eax不确定是否正在使用
          __asm
          {
            cli // 屏蔽中断，不接收CPU信号
            mov eax, cr0
            and eax, not 10000h
            mov cr0, eax
          }

          // 存在内存不可以写
          RtlCopyMemory(pAddress, pBuf, uLength);

          __asm
          {
            mov eax, cr0
            or eax, 10000h
            mov cr0, eax
            sti
          }
          */

          // 方法2：通过虚拟地址计算出物理地址
          pa = MmGetPhysicalAddress(pAddress);
          // 将物理地址映射到内核的内存中，不要缓存，内存属性一般为可读可写可执行
          lpMapBuf = MmMapIoSpace(pa, 0x1000, MmNonCached);
          if (NULL != lpMapBuf) {
            RtlCopyMemory(lpMapBuf, pBuf, uLength);
            // 取消映射
            MmUnmapIoSpace(lpMapBuf, 0x1000);
          }
          Status = STATUS_SUCCESS;
        }

        //MmMapIoSpace()

        // 进程切换回来
        KeUnstackDetachProcess(&ApcState);
```

```
      }
      else {
        dprintf("[51asm] %s pBuf or  Process Invalid\n", __FUNCTION__);
      }
    }
    else {
      dprintf("[51asm] %s %d %s PsLookupProcessByProcessId:%d Error\n",
        __FILE__, __LINE__, __FUNCTION__, UniqueProcess);
    }
  }
  __except (1) {
    dprintf("[51asm] %s %d %s __except\n",
      __FILE__, __LINE__, __FUNCTION__);
  }

  return Status;
}

void DriverUnload(struct _DRIVER_OBJECT* DriverObject) {
  PAGED_CODE();

  UNREFERENCED_PARAMETER(DriverObject);

  dprintf("[51asm]  DriverUnload\n");
}

NTSTATUS DriverEntry(
  __in struct _DRIVER_OBJECT* DriverObject,
  __in PUNICODE_STRING  RegistryPath)
{
  PAGED_CODE();
  UNREFERENCED_PARAMETER(DriverObject);
  UNREFERENCED_PARAMETER(RegistryPath);
  NTSTATUS Status;

  //KdDisableDebugger(); // 限制内核调试器

  dprintf("[51asm] DriverEntry\n");

  unsigned char ch = 0xcc;
  //MyReadProcessMemory((HANDLE)1332, (void*)0x004153b7, (void*)&ch, 1);
  Status =  ((HANDLE)0, (void*)0x004153b7, (void*)&ch, 1);
  if (NT_SUCCESS(Status)) {
    dprintf("[51asm] MyReadProcessMemory ch:%02X\n", ch);
  }
  dprintf("[51asm] MyReadProcessMemory Status:%08X\n", Status);
```

```
    DriverObject->DriverUnload = DriverUnload;
    return 0;
}
```

内核反调试方法

1. EPROCESS --> DebugPoint 调试端口
   目标进程不能调试时，定位该进程的EPROCESS，查看该标志是否为null。
2. KdDisableDebugger（导出）、KdDebuggerEnabled，用来检测内核调试器（Ring3可以继续使用），对于反反调试，可在该函数下断点，查看软件中哪些位置进行了调用（需要使用ART工具提前进行扫描）。
   测试：在驱动代码中调用 KdDisableDebugger 函数，在被调试的操作系统中安装、运行驱动，Windbg调试器不能继续使用。

```
/*
 * @implemented
 */
NTSTATUS
NTAPI
KdDisableDebugger(VOID)
{
    KIRQL OldIrql;

    /* Raise IRQL */
    KeRaiseIrql(DISPATCH_LEVEL, &OldIrql);

    /* TODO: Disable any breakpoints */

    /* Disable the Debugger */
    KdDebuggerEnabled = FALSE;
    SharedUserData->KdDebuggerEnabled = FALSE;

    /* Lower the IRQL */
    KeLowerIrql(OldIrql);

    /* Return success */
    return STATUS_SUCCESS;
} - end KdDisableDebugger -

NTSTATUS
NTAPI
KdEnableDebuggerWithLock(IN BOOLEAN NeedLock)
{
    return STATUS_ACCESS_DENIED;
}
```

```
kd> e 83f29846 c3
kd> u KdDisableDebugger
nt!KdDisableDebugger:
83f29846 c3              ret
83f29847 01e8            add     eax,ebp
83f29849 06              push    es
83f2984a ff              ???
83f2984b ff              ???
83f2984c ffc3            inc     ebx
83f2984e cc              int     3
83f2984f cc              int     3
```

SSDT | ShadowSSDT | FSD | 键盘 | I8042prt | 鼠标 | Partmgr | Disk | Atapi | Acpi | Scsi | 内核钩子 | Object钩子 | 系统中断表

挂钩对象	挂钩位置	钩子类型	挂钩处当前值	挂钩处原始值
len(1) RtlPrefetchMemoryNonTemporal[ntkrnlpa.exe]	[0x83E86E68]->[-]	Inline	90	C3
len(1) KdDisableDebugger[ntkrnlpa.exe]	[0x83F29846]->[-]	Inline	C3	6A
len(4) NtDuplicateObject[ntkrnlpa.exe]	[0x8407F54D]->[0x913021 7E][C:\Users\le...	Inline	2D 2C 28 0D	C3 F8 FE FF
len(1) KiFastCallEntry[ntkrnlpa.exe]	[0x83E8A339]->[-]	Inline	06	05
len(4) NtTerminateProcess[ntkrnlpa.exe]	[0x840A8A02]->[0x913024B2][C:\Users\le...	Inline	AC 9A 25 0D	0E 64 FC FF
len(4) NtTerminateThread[ntkrnlpa.exe]	[0x840C6385]->[0x913024B2][C:\Users\le...	Inline	29 C1 23 0D	8B 8A FA FF
len(1) [ntkrnlpa.exe]	[0x83EC395F]->[-]	Inline	21	01
len(1) [ntkrnlpa.exe]	[0x83EC3BAD]->[-]	Inline	21	01
len(22) [ntkrnlpa.exe]	[0x83EC3D52]->[-]	Inline	E0 0F BA F0 07 73 09...	D8 0F 22 D8 C3 0F 20...
len(1) [ntkrnlpa.exe]	[0x83EC3D6F]->[-]	Inline	00	C3
len(4) [ntkrnlpa.exe]	[0x840ABD1A]->[0x91301F88][C:\Users\le...	Inline	6A 62 25 0D	46 D9 FE FF
len(4) [ntkrnlpa.exe]	[0x840C409A]->[0x91301F88][C:\Users\le...	Inline	EA DE 23 0D	C6 55 FD FF
len(5) [win32k.sys]	[0x9475BC29]->[0x91301DEE][C:\Users\le...	Inline	E9 C0 61 BA FC	8B FF 55 8B EC
len(28) [peauth.sys]	[0x956AAC9D]->[-]	Inline	1E 9D 2B 48 9B 23 A8...	95 27 DA BD 3A 76 0...
len(28) [peauth.sys]	[0x956AACC1]->[-]	Inline	1E 9D 2B 48 9B 23 A8...	95 27 DA BD 3A 76 0...

先在对应的地址上恢复原来的值，再在对应的地址上下一个内存写入断点，等待软件的再次修改，再次修改会产生异常，也就定位到了软件产生异常处的代码。

3. IsDebuggerPresent：Ring3层

```
613: BOOL
614: WINAPI
615: IsDebuggerPresent(VOID)
616: {
617:     return (BOOL)NtCurrentPeb()->BeingDebugged;
618: }
619:
```

```
1072: #define NtCurrentPeb() (NtCurrentTeb()->ProcessEnvironmentBlock)
```

```
25: static inline struct _TEB * NtCurrentTeb(void)
26: {
27:  struct _TEB * pTeb;
28:
29:  /* FIXME: instead of hardcoded offsets, use offsetof() - if possible */
30:  __asm__ __volatile__
31:  (
32:   "movl %%fs:0x18, %0\n" /* fs:18h == Teb->Tib.Self */
33:   : "=r" (pTeb) /* can't have two memory operands */
34:   : /* no inputs */
35:  );
36:
37:  return pTeb;
38: }
39:
```

Win7：Ring0 FS--0x30，Ring3 FS--0x3B。

4. CheckRemoteDebuggerPresent：用于检查远程调试器

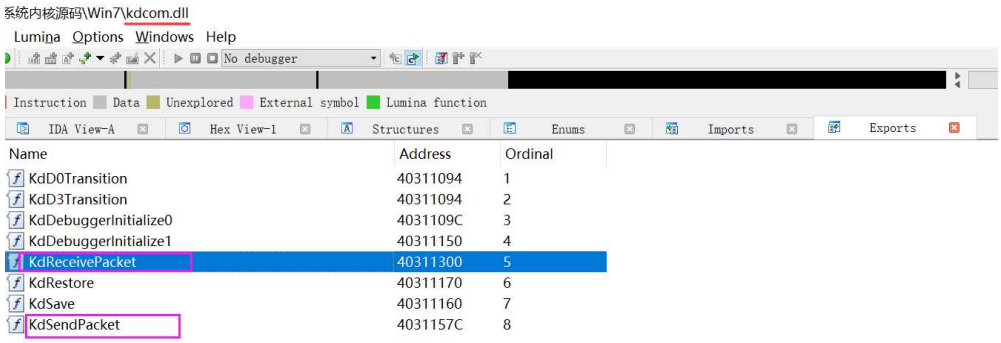
```
406: /*
407:  * @implemented
408:  */
409: BOOL
410: WINAPI
411: CheckRemoteDebuggerPresent(IN HANDLE hProcess,
412:                            OUT PBOOL pbDebuggerPresent)
413: {
414:     HANDLE DebugPort;
415:     NTSTATUS Status;
416:
417:     /* Make sure we have an output and process*/
418:     if (!(pbDebuggerPresent) || !(hProcess))
419:     {
420:         /* Fail */
421:         SetLastError(ERROR_INVALID_PARAMETER);
422:         return FALSE;
423:     }
424:
425:     /* Check if the process has a debug object/port */
426:     Status = NtQueryInformationProcess(hProcess,
427:                                        ProcessDebugPort,
428:                                        &DebugPort,
429:                                        sizeof(DebugPort),
430:                                        NULL);
431:     if (NT_SUCCESS(Status))
432:     {
433:         /* Return the current state */
434:         *pbDebuggerPresent = DebugPort != NULL;
435:         return TRUE;
436:     }
```

NtQueryInformationProcess：

```
318:          /* Process Debug Port */
319:          case ProcessDebugPort:
320:
321:              if (ProcessInformationLength != sizeof(HANDLE))
322:              {
323:                  Status = STATUS_INFO_LENGTH_MISMATCH;
324:                  break;
325:              }
326:
327:              /* Set return length */
328:              Length = sizeof(HANDLE);
329:
330:              /* Reference the process */
331:              Status = ObReferenceObjectByHandle(ProcessHandle,
332:                                                 PROCESS_QUERY_INFORMATION,
333:                                                 PsProcessType,
334:                                                 PreviousMode,
335:                                                 (PVOID*)&Process,
336:                                                 NULL);
337:              if (!NT_SUCCESS(Status)) break;
338:
339:              /* Protect write with SEH */
340:              _SEH2_TRY
341:              {
342:                  /* Return whether or not we have a debug port */
343:                  *(PHANDLE)ProcessInformation = (Process->DebugPort ?
344:                                                  (HANDLE)-1 : NULL);
345:              }
346:              _SEH2_EXCEPT(EXCEPTION_EXECUTE_HANDLER)
347:              {
348:                  /* Get exception code */
```

查询 EPROCESS 的 DebugPort

5. Hook调试器API：
   NtOpenThread
   NtOpenProcess
   KiAttachProcess
   NtReadVirtualMemory
   NtWriteVirtualMemory
   KdReceivePacket、KdSendPacket（kdcom.dll）：双击调试时，负责串口数据的接收、转发。当这两个函数被Hook后，Windbg就接收不到调试事件。



内核中进程隐藏，将进程链表断掉（跳过目标进程）。

系统启动的过程

预引导阶段

1. 系统加点自检，同时完成硬件设备的枚举和配置。
2. BIOS确定引导设备位置，加载引导设备的MBR（引导扇区）。磁盘第一扇区（512字节），用来存放操作系统的启动代码。
3. 在MBR中扫描分区表，定位活动分区，并加载（loader）活动分区上引导扇区到内存。

4. 加载系统根目录的ntldr。

引导过程

1. 初始化Ntldr，完成处理器模式切换和文件系统驱动的加载，如果使用SCSI设备，Ntldr将tbootdd.sys加载到内存。
2. Ntldr读取系统根目录的boot.ini，在屏幕显示系统启动菜单，等待用户选择所需要加载的操作系统。
3. Ntldr读取并运行程序Ntdetect.com，完成硬件的检测。
4. Ntldr根据用户的选择调用系统的硬件配置文件。

内核加载

1. 加载执行体ntoskrnl.exe
2. 加载Hal.dll
3. 加载%systemroot\System32\Config\System下的注册表项HKEY_LOCAL_MACHINE\SYSTEM
4. 选择加载控制集，初始化计算机
5. 根据控制集加载低级硬件设备驱动程序

内核初始化，显示图形界面

1. 内核会使用检测到的硬件数据，在注册表中创建HKEY_LOCAL_MACHINE\HARDWA项
2. 其次的工作是内核通过复制HKEY_LOCAL_MACHINE\SYSTEM\Select子键Current项引用的控制集创建Clone控制集
3. 内核开始进一步加载和初始化设备驱动程序
4. Session Manager（Smss.exe）按顺序启动更高一层次的子系统和各项服务

系统登陆过程

1. 系统首先启动Winlogon.exe
2. 启动Local Security Authority(Lsass.exe)，屏幕显示出登陆对话框
3. 系统执行 Service Controller(Screg.exe)
   再次扫描注册表
   HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control 项并自动加载
   其中系统的或用户的服务
4. 此时，用户已成功的登陆到系统，系统随后把Clone控制集拷贝到
   LastKnownGood控制集

内核初始化

> 同过reactos分析以下函数的实现过程：

KiInitializeKernel

- KiInitSystem
- ExpInitializeExecutive

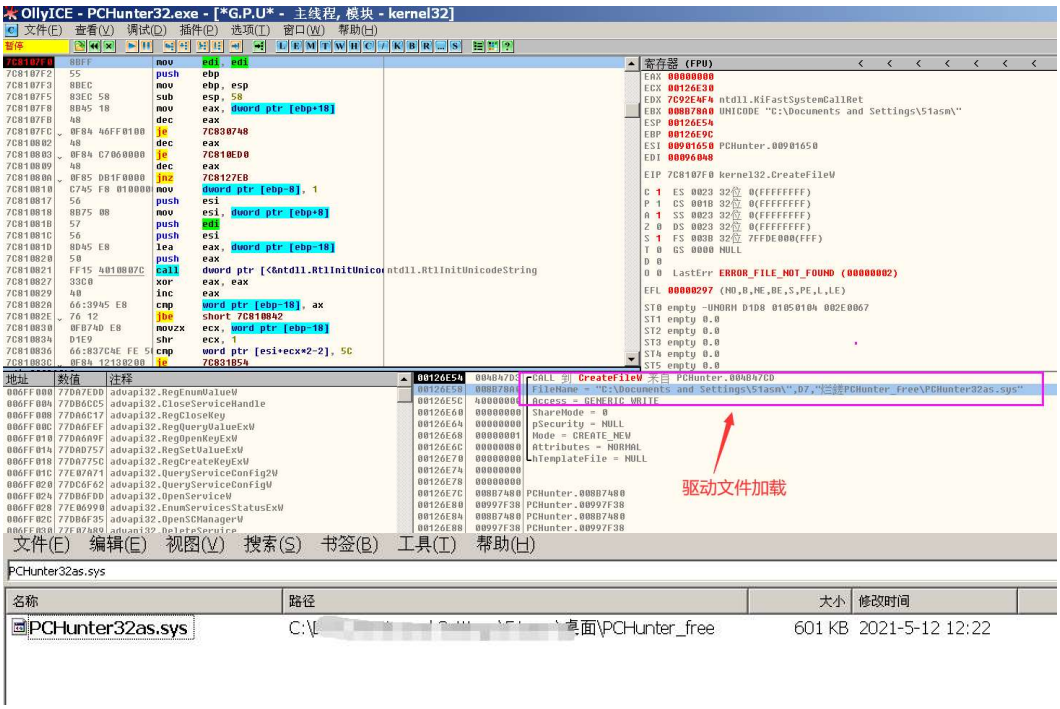
内核漏洞的利用

Ring3中的内核程序都是通过利用 "ntdll.dll" 中未公开的导出函数进入内核或者获取SSDT中的内核函数（通过下标。仅限Zw、Nt开头的函数）。
ZwQueryInformation_xxx
NtQueryInformation_xxx

全局描述表物理地址：0x3F000

驱动的逆向

调试程序，其加载驱动，会通过CreateFileA/CreateFileW创建文件，分析其文件可通过在对应的函数下断点，等待目标文件创建。文件创建后即可将其拷贝走进行分析。



API 监控工具

- api monitor

开源ARK工具源码

- A盾
- OpenARK

EPROCESS 存储到Ring0中，teb、peb存储在Ring3中。在Ring0中，FS指向EPROCESS，在Ring0中，FS指向teb、peb。

创建进程会解析、加载对应的PE文件。