# 高效搜索系统内存检测隐藏进程①

周利荣, 廖建平

(衢州职业技术学院 信息与工程学院, 衢州 324000)

**摘 要**: 分析了进程隐藏方法及常用检测方法, 论述了搜索系统内存检测隐藏进程的原理及实现方法, 即首先判断页面是否有效, 再根据 EPROCESS 结构体特征及 OBJECT 对象头特征来判断内存地址是否为 EPROCESS 地址, 并给出 PAE 内存模式与普通内存模式的判别方法及两种内存模式判断页面是否有效的方法, 探讨了提高搜索效率的方法. 在 windows 7、vista 等操作系统两种内存模式上实验表明可高效枚举所有进程, 包括通过挂钩枚举进程的函数或进入内核空间直接修改内核数据来达到隐藏自身目的的进程.

**关键词**: 进程; 物理地址扩展; 进程环境块; 对象

## Detect Hidden Processes by Searching System Memory with High Efficiency

ZHOU Li-Rong, LIAO Jian-Ping

(Information and Engineering department, Quzhou College of Technology, Quzhou 324000, China)

**Abstract**: The paper analyses the way of hiding processes and common method of detecting hidden Processes and discusses the principle and the way of searching system memory to detect hidden Processes. First judged whether the page is effective or not, Then judged whether memory address is address of eprocess or not according to eprocess's character and object's character. And bring up the way of judging pae memory mode or general memory mode, The way of judging whether the page is effective or not in two memory mode. Discusses the way of improving efficiency. Experiments on windows 7. vista operation system showed that the algorithm can enumerate all processes with high efficiency in two memory mode, These processes hided self by hooking functions, or directly entered into kernel space changed kernel data to hide self.

**Key words**: process; PAE; peb; object

一个进程要运行, 必然会加载到内存中. 基于这个事实, 隐藏进程要在目标机运行, 在内存中一定会存在对应的 EPROCESS 结构. 基于系统内存搜索的进程检测技术利用 EPROCESS 结构体特征找到 EPROCESS 地址指针进而输出进程信息, 可以有效地对进程进行全面的检测. 文献[1]只对普通内存模式(32 位地址总线)的分页处理进行讨论, 未考虑 PAE 内存模式(36 位地址总线)的分页处理, 而现在 32 位处理器采用一种叫 PAE(物理地址扩展)的技术使地址总线增到 36 位, 从而使其方法失去意义. 文献[1]判断内存页所存储的信息是否为 EPROCESS 地址指针的方法是根据 PEB 地址指针前半部分是否相同. 文献[1]将搜索范围定在系统内存 0x80000000 至 0x90000000, 没有理论依据, 实践证明效率低且有漏检进程. 文献[2]考虑了两种内存模式的分页处理情况, 但没有给出判断两种内存模式的实用程序. 文献[2]判断内存页所存储的信息是否为 EPROCESS 地址指针的方法是根据对象头(Object Header)类型值是否相同, 该方法不适用于 windows 7 等最新操作系统. 文献[2]将搜索范围定在系统内存 0x80000000 至 0x9FFFFFFF、0xFB000000 至

① 收稿时间:2012-02-13;收到修改稿时间:2012-04-10

0xFFDFEFFF、0xFFDFF000 至 0xFFFFFFFF, 没有理论依据且实践证明效率低且检测也不完整. 本文考虑了两种内存模式的分页处理情况且对 PDE、PTE 指针公式进行推导. 判断内存页所存储的信息是否为 EPROCESS 地址指针的方法是根据 PEB 地址指针前半部分是否相同和对象头(Object Header)类型值是否相同, 实践证明搜索更可靠. 本文对 x86 系统地址空间分布规律进行深入研究, 表明进程的 Eprocess 结构体就在系统内存的非分页缓冲池中, 并根据内核变量 MmNonPoolStar、MmSizeOfNonPagedPoolInBytes、MmNonPagedPoolExpansionStart、MmNonPagedPoolEnd 进一步缩小了搜索范围. 并对常用隐藏进程检测方法检测效率及可靠性进行了分析.

## 1 进程隐藏方法

(1) 在用户态挂钩或修改枚举进程的 API 如 ZwQuerySystemInformation 函数, 可以改变系统函数的行为从而实现进程的隐藏, 让它们返回的数据始终“遗漏”病毒进程自身的信息.

(2) 在内核态更改系统内核中的数据结构, 直接隐藏某些对象. 每一个进程都是由 EPROCESS 结构体来描述的, 所有进程的结构体链接成一个环形的双向链表. 隐藏进程的一种办法就是进入内核空间, 这样它就拥有了和内核一样的访问权限, 修改链表中的指针, 使在枚举进程时能绕过要隐藏的进程.

(3) 在内核态挂钩系统服务描述表(System Service Dispatch Table, SSDT). Windows 操作系统中所有的上层 API 函数, 在系统的内核都有对应的系统服务函数, 从用户态函数转到内核态函数的对应关系就是依靠 SSDT 表来实现的, 当挂钩 SSDT 后, 进程枚举查询类函数对应的内核服务函数地址在 SSDT 中被修改后, 原函数运行被劫持. 原函数所得结果也会被修改, 最终使用户不能获得所有进程的相关信息.

(4) 挂钩中断调度表(Interrupt Dispatch Table, IDT). 在 Windows 操作系统中, 所有的用户态 API 函数都是通过一个特殊的软中断进入到内核的, 一旦软中断出现, 系统从 IDT 中调用相应的中断服务处理例程, 进入到内核模式. 通过对 IDT 表的挂钩, 我们也可以实现对系统所有函数调用的监视和修改.

## 2 基于系统内存搜索的进程检测技术的实现方法

EPROCESS 结构体就在系统内存的非分页缓冲池中, 可根据非分页池的空间确定间确定搜索范围, 首先判断内存是否开启 PAE 模式及相应内存模式的页面是否有效. 再判断该页所存储的信息是否为 EPROCESS 地址指针. 最后判断进程是否结束, 如果是非结束输出进程信息.

### 2.1 程序由用户态进入内核态的方法

无论是确定内存模式还是判断页面是否有效程序首先需由用户态进入内核态.

(1) 可以编写驱动程序进入内核. 当驱动程序启动时, 操作系统会首先调用 DriverEntry() 函数, DriverEntry()函数是驱动程序的入口点. DriverEntry()函数又调用 PsCreateSystemThread()函数在内核中创建线程, 其功能是搜索系统内存检测进程.

(2) 利用调用门进入内核. 在系统的全局描述符表(GDT 表)中建立一个调用门描述符, 然后应用程序通过这个调用门发起调用, 从而在 Ring0 下执行自己的代码, 这样应用程序便具有了特权. [1]

### 2.2 PAE 内存模式与普通内存模式的区别及判定

由于在 32 位处理器架构下, 对内存的访问限制在 4GB 以下的空间. 为了突破 4GB 的限制, 现在的 32 位处理器采用一种叫 PAE（物理地址扩展）的技术, 来实现对超出 4GB 空间的物理地址的访问. PAE 实际上采用了 36 位的地址总线, 这样理论上可以支持 64GB 内存空间的寻址. 确定内存模式有两种方法:

(1) 从 MmIsAddressValid 中查询内存模式

MmIsAddressValid 函数的功能是判断地址是否有效, 在 PAE 内存模式与普通内存模式其代码是不相同的, 用 windows 内核调试工具 windbg 的命令 “u MmIsAddressValid l 50”可显示汇编代码, windows 7 系统 PAE 内存模式代码为:

```
83c4faa9  8bff          mov     edi,edi
83c4faab  55            push    ebp
83c4faac  8bec          mov     ebp,esp
83c4faae  8b5508        mov     edx,dword ptr [ebp+8]
……
```

windows 7 系统非 PAE 内存模式代码为:

```
8400c180  8bff          mov     edi,edi
```

```
8400c182  55          push   ebp
8400c183  8bec        mov    ebp,esp
8400c185  8b4d08      mov    ecx,dword ptr [ebp+8]
……
```

可知 PAE 内存模式第四指令为“8b5508”，而非 PAE 内存模式第四指令为“8b4d08”，可据此编写自定义函数 HowPaging()区分内存模式.

```
int HowPaging()
{ RtlInitUnicodeString(&ustr, L"MmIsAddress Valid");
i = (ULONG)MmGetSystemRoutine Address (&ustr);
for(end = i + 40; i < end; i++)  // 只搜索前 40 个字节
{ if(*(PUCHAR)i==0x8b & &
*(PUCHAR)(i+2) )==0x08)
{ if(*(PUCHAR)(i + 1) == 0x4d){ return 0; }
  else if(*(PCHAR)(i + 1) == 0x55)
  { return 1; } }}}
```

不同操作系统的 MmIsAddressValid 函数代码不相同，因此此方法不适用于所有操作系统.

(2) 控制寄存器 CR4 的第 5 位标记是否开启 PAE

自定义函数 cr4()返回 CR4 的第 5 位标记值. VC++内联汇编语句 mov eax,CR4; mov uCR4, eax; 可将控制寄存器 CR4 的值保存在变量 uCR4 中，将 uCR4 与 0x20 相与后再右移 5 位即得 CR4 的第 5 位标记值，若为 1 则表示开启 PAE，为 0 表示未开启 PAE. 其代码如下:

```
Int cr4( )
{int uCR,uCR4 ;
__asm
{_emit 0x0F
_emit 0x20
_emit 0xE0 //_EMIT 伪指令相当于 MASM 中的 DB,但一次只能定义一个字节，mov eax,CR4 就是 0F 20 E0
mov uCR4, eax }
uCR=(uCR4&0x20)>>5; return uCR; }
```

### 2.3 普通内存模式判断页面是否有效的方法

x86 将物理地址空间分为 4KB 的小块，称为内存页. 如此算来，4GB 的内存就有 1024x1024 个内存页了. 非 PAE 模式，处理器使用二级索引结构来管理内存页，一维叫做页目录(Page Directory)，二维叫做页表(Page Table)，一个页目录包含 1024 项，称为页目录项 PDE (Page Directory Entry)，每个页目录项指向一个页表，这样我们就有 1024 个页表. 每个页表也有 1024 项，称为页表项 PTE(Page Table Entry)，每个页表项指向 4KB 页.

如果我们访问的虚拟地址所在页在物理内存里，虚拟地址所在页相应的 PDE、PTE 就都是有效的，将其交给 CPU 转换成物理地址后访问就可以了. 如果页不在物理内存中，那对应的 PDE，PTE 都是无效的. 逻辑地址到物理地址的转换是由处理器完成的. 非 PAE 模式，一个 32 位的逻辑地址被分成下图所示的 3 部分。

| 10bits | 10bits | 12bits |
|---|---|---|

处理器首先根据 CR3 找到页目录的物理地址，然后由逻辑地址的前 10 位来索引对应的 PDE，接着的 10 位来索引页表的 PTE，从而得到 PTE 指向的 4KB 物理内存页，逻辑地址的低 12 位用来指向内存页中具体的位置，相当于页内的一个偏移. 不难得出由虚拟地址 addr 得到 PDE 指针的计算公式为: PDE 指针=页目录基址+[addr>>(页表索引位数+页内偏移)]* PDE 长度. 由虚拟地址 addr 得到 PTE 指针的计算公式为: PTE 指针=页表基址+(addr>>页内偏移)* PTE 长度. 页表索引位数为 10，页内偏移为 12，由 windbg 命令 dt _hardware_pte 可知 PDE、PTE 长度为 32 位即 4 字节. 由 windbg 命令!pte 0(求虚拟地址 0x0 对应的 PDE 和 PTE 的虚拟地址)可知页目录基址为 0xc0300000，页表基址为 0xc0000000. 即 PDE 指针= 0xc0300000 + (addr>>22)*4, PTE 指针=0xc0000000 + (addr >> 12)* 4.

判断 PDE 的 0 位是否为 1，如果不为 1 则 PDE 无效，对应的 1024 个页面不在物理内存中，虚拟地址递增 4Mb (1024*4Kb). 如果 PDE 的 0 位为 1，表示对应的 1024 个页面全部或部分在物理内存中，则判断 PDE 的 7 位是否为 1，如果为 1 则表示对应的 1024 个页面全部在物理内存中. 如果 PDE 的 7 位为 0，表示对应的 1024 个页面中部分页面在物理内存中，则判断 PTE 的 0 位是否为 1，如果为 1 说明 PTE 有效，页在物理内存中，如果为 0 则 PTE 无效，页面不在物理内存中，虚拟地址按 4kb 步进.

### 2.4 PAE 内存模式判断页面是否有效的方法

PAE 模式, 将原来的使用二级索引结构扩展为三级索引结构, 也就是在原来的页目录和页表基础上再增加一级, 称为页目录指针表, 每项 64 位. 32 位线性地址被分割为如下三个部分: 2 位(位 30 和位 31)的页目录指针表索引, 用来索引本地址在页目录指针表中的对应表项. 9 位(位 21-29)的页目录表索引, 用来索引本地址在页目录表中的对应表项. 9 位(位 12-20)的页表索引, 用来索引本地址在页表中的对应表项. 12 位(位 0-11)的页内偏移.

| 2bits | 9bits | 9bits | 12bits |
|---|---|---|---|

页表索引位数为 9, 页内偏移为 12. 由 windbg 命令 dt _hardware_pte 可知 PDE、PTE 长度为 64 位即 8 字节. 由 windbg 命令!pte 0(求虚拟地址 0x0 对应的 PDE 和 PTE 的虚拟地址)可知页目录基址为 0xc0600000, 页表基址为 0xc0000000. PDE 指针 =0xc0600000 + (addr>>21)*8, PTE 指针 = 0xc0000000 + (addr >> 12)* 8.[2] 判断 PAE 模式虚拟地址所在页面是否有效的函数如下:

```
PAGE_STATUS CheckPageValid(DWORD addr)
{ pde = 0xc0600000 + (addr>>21)*8;
if( (*(PULONG)pde & 0xl )==0)       return
INVALID_ PDE;
if(  (*(PULONG)pde  &  0xl  )  !  =0  &&
(*(PULONG)pde&
0x80)!=0)        return VALID_PAGE;
if(  (*(PULONG)pde  &  0xl  )  !  =0  &&
(*(PULONG)pde&
0x80) == 0)
{pte = 0xc0000000 + ( addr >> 12 ) * 8 ;
return ( (*(PULONG)pte & 0xl) !=0) ?
VALID_PAGE:INVALID_PTE ;  }    }
```

### 2.5 判断该页所存储的信息是否为 EPROCESS 地址指针的方法

当获取内存页以后, 就需要判断该页所存储的信息是否为 EPROCESS 地址指针, 须满足两个条件.

(1) 每个进程的 EPROCESS 结构体的 PEB 变量成员保存着 PEB 地址指针, 而 PEB 地址指针前半部分都相同, 均为“0x7FFD”, 所以可以通过比较 PEB 地址指针的前半部分来判断是否为 EPROCESS 地址指针.

(2) Windows 系统的各种资源以对象(Object)的形式来组织, 而各种 Object 的共有的信息(对象类型、对象的引用计数、句柄数等信息)保存在 OBJECT_HEADER 与其他的几个结构中. 换而言之, 在对象管理器内部, 不同类型的对象具有相同的对象头(Object Header), 但 Object Body 部分却是不同的.

Object Header 结构体如下:

```
+0x000 PointerCount     : Int4B
+0x004 HandleCount    :Int4B
+0x004 NextToFree       : Ptr32 Void
+0x008 Type          : Ptr32 _OBJECT_TYPE
+0x00c NameInfoOffset     : UChar
……
```

Object Header 偏移 0x008 处 Type 成员为对象类型值,相同类型的对象具有相同的值. 自 Window 7 开始, _OBJECT_HEADER 及其之前的一些结构发生了变化.

```
lkd> dt _object_header
nt!_OBJECT_HEADER
+0x000 PointerCount         : Int4B
+0x004 HandleCount          :Int4B
+0x004 NextToFree           : Ptr32 Void
+0x008 Lock                 : _EX_PUSH_LOCK
+0x00c TypeIndex            : UChar
……
```

可以看到, 0x008 处的指向_OBJECT_TYPE 的指针已经没有了, 取而代之的是在 0x00c 处的类型索引值. 但 Windows7 中添加了一个函数 ObGetObjectType, 返回 Object_type 对象指针.

### 2.6 提高搜索效率的方法

32 位 Windows 系统允许每个进程拥有自己的 4GB 逻辑地址空间, 即 0x00000000 到 0xFFFFFFFF. 4G 的逻辑地址空间中, 可供用户操作的只有低位的 2GB(是在用户模式下可操控的), 高位的 2GB 是 windows 核心模式使用的, 是内核态的地址空间, 即 0x80000000 到 0xFFFFFFFF, 这里存放着整个内核的代码和所有的内核模块, 以及内核所维护的数据,包括 EPROCESS 结构体. 因此搜索范围初步确定为 0x80000000 到 0xFFFFFFFF 的 2G 空间.

在 2G 系统内存搜索 Eprocess 结构体效率低, 因此必须清楚 x86 系统地址空间分布规律, 缩小搜索范围. x86 系统地址空间分布如下:

(1) 0x80000000--0x9FFFFFFF: 引导系统(Ntoskrnl.exe 和 Hal.dll)和非分页缓冲池初始部分的系统代码.

(2) 0xA0000000--0xA3FFFFFF: 系统映射视图(如 Win32k.sys)或者会话空间.

(3) 0xA4000000--0xBFFFFFFF: 附加系统页表项(PTE)或附加系统高速缓存.

(4) 0xC0000000--0xC03FFFFF: 进程页表和页目录, 描述虚拟地址映射的结构.

(5) 0xC0400000--0xC07FFFFF: 超空间和进程工作集列表.

(6) 0xC0800000--0xC0BFFFFF: 未使用区域, 不可访问.

(7) 0xC0C00000--0xC0FFFFFF: 系统工作集链表, 描述系统工作集的工作集链表数据结构.

(8) 0xC1000000--0xE0FFFFFF: 系统高速缓存, 用来映射在系统高速缓存中打开的文件的虚拟空间.

(9) 0xE1000000--0xEAFFFFFF: 分页缓冲池, 可分页系统内存堆.

(10) 0xEB000000--0xFFBDFFFF: 系统页表项和非分页缓冲池.

(11) 0xFFBE0000--0xFFFFFFFF: 系统性故障转储信息和硬件抽象层(HAL)使用区域.

分页缓冲池上分配的内存可以交换到虚拟内存, 当程序需要这些页面的时候, 再读到内存. 非分页缓冲池里分配的内存是不能交换到虚拟内存上面的, 假如放到分页缓冲池并被交换到磁盘上时可能会发生灾难性的后果, 进程的 EPROCESS 结构体就在非分页缓冲池中. 因此进程的 EPROCESS 结构体在(1)0x80000000--0x9FFFFFFF(引导系统(Ntoskrnl.exe 和 Hal.dll)和非分页缓冲池初始部分的系统代码.)及(10)0xEB0000000--0xFFBDFFFF(系统页表项和非分页缓冲池)两块区域. 这样大缩小了搜索范围.

(1) 0x80000000--0x9FFFFFFF 还包含了引导系统(Ntoskrnl.exe 和 Hal.dll),(10) 0xEB0000000--0xFFBDFFFF 还包含了系统页表项. 若能将这部分内存空间剔除, 能进一步缩小搜索范围. 而非分页池的空间由两块区域组成, 常规的区域是已经申请好物理内存的从内核变量 MmNonPagedPoolStart 开始, MmNonPagedPoolStar+MmSizeOfNonPagedPoolInBytes 为结束,其中内核变量 MmSizeOfNonPagedPoolInBytes 为非换页内存池的大小. 另外含一块扩展区域, 由内核变量 MmNonPagedPoolExpansionStart 与内核变量 MmNonPagedPoolEnd 指定, 只有在内存不够时采取分配申请和释放的操作.

内核中 FS 寄存器指向 KPCR(内核处理器控制域)的结构, 这是一个很有用的结构, 偏移送 0x34 处有 KdVersionBlock 变量, KdVersionBlock 对应的结构体应该是 _DBGKD_GET_VERSION64, 其中变量 DebuggerDataList 对应的结构体应该是 _KDDEBUGGER_DATA64, 其中包含 MmNonPagedPoolStar、MmNonPagedPoolEnd 等内核变量. 部分代码如下:

```
_asm
{ mov eax,fs:[0x01C]
mov eax,[eax+0x34]
mov KdVersionBlock,eax  }
pKdData64=(PKDDEBUGGER_DATA64)*((PULONG)KdVersionBlock->DebuggerDataList);
```

则 *(PULONG)pKdData64->MmNonPagedPoolStart 为非分页池空间的常规区域起始值.

在测试系统 Windows 7 中, 经过多次的实验, 将检测结果与检测耗时汇总统计如表 1 所示.

表 1 三种搜索范围的检测时间及检测全面性比较

| 搜索范围 | 时间(毫秒) | 是否有漏检进程 |
|---|---|---|
| 0x80000000--0x9FFFFFFF | 2.3~2.5s | 有 |
| 0x80000000至x9FFFFFFF、0xFB000000至xFFDFEFFF、0xFFDFF000至xFFFFFFFF | 2.6~2.7s | 有 |
| 改进后本方法 | <0.5s | 没有 |

## 3 常用隐藏进程检测方法检测技术效率及可靠性分析

常用隐藏进程检测方法有: (1) 直接遍历 EPROCESS 双向链表检测隐藏进程; (2) 枚举 PspCidTable 表检测隐藏进程; (3) 拦截系统调用的隐藏进程技术; (4) 挂钩 SwapContext 检测隐藏进程; (5) 挂接系统服务调度表的隐藏进程检测技术; (6) 本文讨论的基于内存搜索的进程检测技术.

方法(1)和(2)由于对 Eprocess 结构体进行组织, 所以检索效率较高. 方法(3)原理是当进程转向系统调用接口时将其拦截, 由于进程可能长时间处于等待状态

并且不进行系统调用, 因此效率不高. 方法(4)是在每次线程切换时定位线程, 进而根据线程找到该线程的进程进行对比以得到真正的进程列表. 如果进程在休息, 没有线程调度发生, 也就检测不到进程, 其效率不高.

方法(5)是将系统服务调度表中相应的函数指针重新定位, 使它指向我们自己定义的函数. 这样, 当进行系统服务调用时, 就可以通过自己定义的函数进行完预处理之后, 再调用原来的系统服务函数, 从而避免被恶意软件拦截并删除自身进程信息, 此方法效率较高. 方法(6)如果在 2G 系统内存范围搜索, 其效率是极低的, 根据本文的方法缩小搜索范围后, 效率与方法(1)(2)相当.

方法(1)可被病毒进程轻易修改双向链表中的指针, 使进程从链表中脱离, 在枚举进程时就能绕过要隐藏的进程, 因此可靠性不高. 方法(2)枚举 PspCidTable 依靠于 PspCidTable 中已经能够被恶意程序任意修改的句柄类型信息, 使得该检测技术失效. 方法(3)挂钩系统调用的技术目前已经能够被修改执行系统调用的中断技术或是用 GDT 中的调用门实现系统调用的技术所避开, 可靠性也不高. 方法(4)挂钩 SwapContext 检测隐藏进程在每次线程切换时就能定位线程, 从而根据线程找到进程, 可靠性较高. 方法(5)条件是恶意软件程序是通过拦截 Advapi32. dll 中的 OpenProcess 函数的返回结果隐藏自身进程, 因此可靠性不高.[3]

既然搜索系统内存检测隐藏进程的原理是根据 EPROCESS 结构体的 PEB 特征值及对象头类型值, 只要修改进程的 PEB 特征值及对象头类型值即可逃避检测. 如进程“winword.exe”的 EPROCESS 地址为 0x8659c468, windows 7 系统 eprocess 结构体的 PEB 偏移为 0x1a8, 所以 0x8659c468+0x1a8=0x8659c610 为进程“winword.exe”的 PEB 地址.

lkd>dd 8659c610，结果为:

659c610 7ffd3000 00000000 0000011a 00000000,

其高字节“7ffd”为 PEB 特征值, 使用 windbg 命令 eb 修改其值,

Lkd>eb 8659c610 41 41 30 00

Lkd>dd 8659c610 结果为:

8659c610 00304141 00000000 0000011a 00000000

进程“winword.exe”的 PEB 特征值已修改, 再执行搜索系统内存检测隐藏进程的程序发现没有“winword.exe” 进程, 但用 word 编辑文本则提示应用程序非正常关闭. 对系统进程“wininit.exe”进行同样修改发现检测不到进程, 但系统随即蓝屏崩溃. 对于基于内存搜索的隐藏进程检测技术来说, 修改 EPROCESS 的结构特征值的同时要保证进程的运行在实现上是不可能的, 因此该技术在目前能够检测到各种方法隐藏的进程, 具有很好的可靠性.

将实验所用恶意代码 A(NtIllusion)、B(Klog)、C(BootRootkit)、D(FUTo)、E(FU)、F(Patchfinder2)、G(Vanquish)分别安装在测试系统 Windows 7 中, 然后分别使用现有的检测技术, 经过多次的实验, 将检测结果与检测耗时汇总统计如表 2 所示.

表 2 六种检测技术的可靠性及效率比较

| 检测方法 / 恶意代码 | (1) | (2) | (3) | (4) | (5) | (6) |
|---|---|---|---|---|---|---|
| A | Y | Y | Y | Y | Y | Y |
| B | Y | Y | Y | Y | Y | Y |
| C | | Y | Y | Y | Y | Y |
| D | | Y | Y | Y | | Y |
| E | | Y | | Y | | Y |
| F | | | | Y | | Y |
| G | | | | Y | | Y |
| 耗时 | <0.5s | <0.7s | 5~7s | 15~20s | 2.9s | <0.5s |

## 4 结语

本文考虑了两种内存模式的分页处理情况, 对内核态的地址空间分布规律进行深入研究, 进一步缩小了根据 EPROCESS 结构体特征和对象头(Object Header)类型值特征搜索 EPROCESS 结构体的范围, 提高了搜索效率, 并对常用隐藏进程检测方法检测效率及可靠性进行了分析.

## 参考文献

1 潘茂如,曹天杰.基于直接操作内核对象的进程隐藏技术研究.计算机工程,2010,36(18):138–140.

2 王璟,武东英.基于内存扫描的隐藏进程检测技术.计算机应用,2009,29(6):89–91.

3 胡和君,范明钰.基于内存搜索的隐藏进程检测技术.计算机应用,2009,29(1):175–177,188.