ProcL - Detecting Hidden Processes

(Scanit R&D Labs)

Abstract

In this paper, we introduce different approaches for detecting hidden processes on Windows platform. A hidden process is a process, which executes as a normal process but in a manner that is unobservable by many of the operating system's reporting functions. We have implemented a mechanism to combine different approaches that detects and differentiates hidden processes.

Introduction

Rootkit can be difficult to detect, especially when they are running in kernel. And therefore more difficult to prevent against. This is because they are running into kernel, they can alter functions used by all applications running on the system. These applications will include antivirus, anti-spyware, anti-rootkit etc. Whatever changes made by anti-rootkit or rootkit detectors to prevent against rootkit can simply be unblocked by the better rootkit. The same powers are available with infectors and preventers. This does not mean that all is lost for preventers. But one thing has to be always on the mind of detectors/preventers that what works today, may not work tomorrow.

One of the inherent and malicious act of any rootkit is to either hide itself or hide the victim process. Consequences of hiding processes is that even legitimate copies of system utilities will fail to list information about the processes executing on the system. The inherent danger of a hidden process is that a preventing system that functions under the assumption that the underlying system is operating according to the system specification, will never see the intrusive processes and will give the system owner the false sense of confidence that the system is normal and hence secure.

Hiding a process is particularly threatening because it represents some malicious code running on your system that you are completely unaware of. Process hiding has a significant effect. Many of the trojan, virus, spyware, rootkit writers use similar techniques to hide themselves and stay undetected as long as possible on target machines. Finding all the ways a rootkit might hide a process is just the first step in defending against the rootkits. Detecting hidden objects is a promising new area in rootkit detection.

It is necessary to have protection against the hidden processes, if you want to stay secured. Many of the antivirus and antispyware manufacturing companies falling back as they are not able to come up with any solutions for hidden processes. There are only few tools which can detect hidden processes, but are you willing to pay them considerable amount of money?

We believe that trend in attack tools is the continued advancements of the means to hide the presence of intrusive processes.

Detecting Hidden Processes (ProcL)

In ProcL we are using different approaches to detect hidden processes. Essentially, we have detailed and implemented a mechanism to embed all these different approaches in one tool to detect hidden processes. Our methods of detecting hidden processes requires the examination of each kernel object - EPROCESS, ETHREADS, HANDLES, JOBS.

ProcL employs many different methods to detect hidden processes. Therefore, we believe, ProcL would defeat process concealment from one certain method.

User-mode Detection Approach (ring-3)

Method 1:

The simplest of all process enumeration methods is enumerating list of processes using ToolHelp API.

```
    hSnapshot   = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if(hSnapshot != INVALID_HANDLE_VALUE)
    {
      PROCESSENTRY32  pEntry  = { sizeof(PROCESSENTRY32) };
      BOOL         bOk     = Process32First(hSnapshot, &pEntry);

      for (; bOk; bOk = Process32Next(hSnapshot, &pEntry))
      {
        printf("%lu\n", pe.th32ProcessID);
      }
    }
```

This methodology may not find any hidden process though. But this can certainly be used as a reference list to differentiate hidden processes found in other methods. We certainly need to maintain a list of all processes we enumerate in any of our methodology to differentiate and identify hidden processes.

Method 2:

This second method is to retrieve the process identifiers of all running processes by calling EnumProcesses API. This API fills an array of DWORD values with the identifiers of all processes in the syatem.

```
  DWORD dwProcesses[1024], sizeofBytes;

  if(EnumProcesses( dwProcesses, sizeof(dwProcesses), &sizeofBytes) )
  DWORD dwCount = sizeofBytes / sizeof(DWORD);

  for (DWORD i = 1; i <= dwCount; i++ )
  {
    printf("%lu\n", dwProcesses[i]);
  }
```

It is unlikly to find anything hidden at this stage. But as mentioned above it is necessary to use all differnet methods to come up with the list of running processes we can check against to differentiate hidden processes. Again, the outcome of this method is going to be used as reference list for other advanced methods.

Method 3:

The next method is to acquire a list of running processes by using an windows undocumented API ZwQuerySystemInformation. A little advance programming skill is required to use this method.

```
  BYTE     *pBuffer= 0;
  ULONG ulLength= 0;
  ULONG    ulStatus= 0;
  ULONG ulOffset= 0;
  PROCESS_SYSTEM  *pProcessSystem
  = NULL;
  ULONG i  = 0;

  do
  {
    if(pBuffer != 0)
      free(pBuffer);

    ulLength += 0x10000;

    if((pBuffer = (BYTE *)malloc(ulLength)) != 0)
      ulStatus = ZwQuerySystemInformation(5
                        /*InfoClass: SystemProcessesAndThreadsInformation*/,
      pBuffer, ulLength, &ulLength);

  } while(ulStatus == 0xC0000004L);
  // STATUS_INFO_LENGTH_MISMATCH

  if(ulStatus == 0)
  {
    pProcessSystem = (PROCESS_SYSTEM *)pBuffer;
    while(ulOffset < ulLength)
    {
      pProcessSystem = (PROCESS_SYSTEM*)(pBuffer + lOffset);

      lOffset += pProcessSystem->NextEntryDelta;
```

```
        if(pProcessSystem->ProcessId == 0)
        {
           printf("%d %d %ls %d %d\n", pProcessSystem->ProcessId,
              pProcessSystem->InheritiedFromProcessId,
              L"System Idle Process",
              pProcessSystem->HandleCount,
              pProcessSystem->ThreadCount);
        }
        else
        {
           printf("%d %d %ls %d %d\n", pProcessSystem->ProcessId,
              pProcessSystem->InheritiedFromProcessId,
              pProcessSystem->ProcessName.Buffer,
              pProcessSystem->HandleCount,
              pProcessSystem->ThreadCount);
        }

        if(pProcessSystem->NextEntryDelta == 0)
           break;
     }
  }
```

I leave it to readers to find out how to use ZwQuerySystemInformation API. API could be found from "NTDLL.DLL". This method may or may not reveal much for us. But it is always good to check it.

Method 4:

The next method is again about using the same undocumented API, ZwQuerySystemInformation. This time we will use this API to list all the open handles. From handle's list we can try to find the processes which has opened these handles and are using them. With the help of advanced programming skill, it is very easy to use this method.

```
  BOOL bShouldContinue = FALSE;
  DWORD   dwSize     = 0X4000;
  ULONGlBufSize   = 0x20000-0x8000;
  PSYSTEM_HANDLE_INFORMATION HandleInfo = NULL;
  LONG lStatus    = 0;

#define SystemHandleInformation 16

#define STATUS_INFO_LENGTH_MISMATCH (LONG(0xC0000004L))

  do
  {
     lBufSize += 0x8000;
     if(HandleInfo)
        VirtualFree(HandleInfo, 0, MEM_RELEASE);
     HandleInfo = (PSYSTEM_HANDLE_INFORMATION)VirtualAlloc(NULL,
                       BufSize,
                       MEM_COMMIT,
                       PAGE_READWRITE);
     if(HandleInfo != NULL)
        Status = NtQuerySystemInformation(SystemHandleInformation,
                 (unsigned long *)HandleInfo,
                 BufSize,
                 NULL);
     else
        break;
  } while(Status == STATUS_INFO_LENGTH_MISMATCH);

  if(Status < 0)
  {
     VirtualFree(HandleInfo, 0, MEM_RELEASE);
     return FALSE;
  }

  DWORD dwNumEntries = HandleInfo->NumberOfHandles;
  DWORD i     = 0;
  DWORD j     = 0;


  for(i=0; iHandles[i].UniqueProcessId)
        printf("%lu\n", HandleInfo->Handles[i].UniqueProcessId);
  }
```

Many rootkits and viruses do not hide handles, opened by the process, when they hide the process. This methodology should definitely find something hidden for us. How much can we rely on this methodology? Well, hiding handles is also be possible while hiding a process. But, I am not sure how many rootkits or viruses are doing that.

Method 5:

This next method we are going to discuss about is a kind of process bruteforcing. Here, you need to iterate through big list of numbers starting from 0x08 to 0xFFFF. Try to find out whether a number is given to any process currently running on the system.

```
DWORD tempPid;
for ( tempPid = 0x08; tempPid < 0xFFFF; tempPid +=4 )
{
     HANDLE hProcess = OpenProcess(  PROCESS_VM_READ | PROCESS_QUERY_INFORMATION
                  FALSE, tempPid );

     if (hProcess)
     {
        printf("%lu\n", tempPid);
        CloseHandle( hProcess );
     }
}
```

This should work against all DKOM methods but it may not work against any kernel mode hooks.

Method 6:

There is another method based on handle enumeration.
Essentially, the method is to enumerate the list of processes by analyzing the handles open in another process related to itself. All processes running in the system have to be started by some means of parent process. Therefore, parent process will have handles of each process it has started, unless parent has closed such handles which is uncommon. Win32 subsystem known as, csrss.exe, will have handles of all available processes. While iterating handle list -
1. For each handle of a process, PID of the process can be found using ZwQueryInformationProcess API.
2. For each handle of a thread, PID of the process can be found using ZwQueryInformationThread API.
3. For each handle of a job, PID of the process can be found using ZwQueryInformationJobObject.

Method 7:

Performance Data Helper (PDH) library available in Windows NT based system is a convenient interface to performance data enumeration. The performance monitoring architecture defined in Windows NT based system delineates a concept of object, for which performance counting is made. Examples of objects are processor, processes, disk drives etc. Each object can have one or more instances along with a set of performance counters. Performance counters are used to provide information as to how well the operating system or an application, process, driver is performaing. Using the value of the performance counter "ID Process", we can enumerate all instances of the object named "Process". PDH APIs are implemented and exported in the PDH.dll file. The working of these approach is based on enumerating the performance counters that we are interested in. Mainly those for processes.

We need to maintain a list of enumerated process and need to flag them as hidden if they are not already found in first 2 methods (ToolHelpAPI and EnumProcess) we discussed above.
Some of the above mentioned methods only allow us to find process identifiers. Using these process identifiers, we will have to enumerate other process related informations such as process name, number of modules + threads + handles being used by the process, command line arguments, if any, of the process etc.

Remember, all of these are user mode methods. You do not need to write any sort of kernel driver to use these methods.

Kerne-mode Detection Approaches (ring-0)

Method 1:

The first method we are going to use here in to acquiring the list of running processes by iterating through the EPROCESS structures.
Each process is represented by an executive process (EPROCESS) block. Besides containing many attributes relating to a process, an EPROCESS block contains and points to a number of other related data structures. For example, each process has one or more threads represented by executive thread (ETHREAD) blocks. The EPROCESS block and its related data structures exist in system space, with the exception of the process environment block (PEB), which exists in the process address space (because it contains information that is modified by user-mode code).

For a list of most of the fields that make up an EPROCESS block and their offsets in hexadecimal, type !processfields in the kernel debugger.
The kernel maintains a circular doubly linked-list of all active processes, known as the ActiveProcessLinks. Each active process is represented on this linked-list with a EPROCESS structure. The exported PsInitialSystemProcess symbol points us to the EPROCESS structure of the System process.
To obtain a name of process, process ID, parent processID and other attributes, use specific offsets from EPROCESS structure. Find more details about EPROCESS offsets from here (http://blog.scanit.net/index.php?/archives/3-Magical-offsets-track-and-analyze-in-memory-kernel-objects-which-represents-process-and-threads.html).

```
void EnumerateProcessList()
{
   PLIST_ENTRY ProcessListHead, ProcessListPtr;
   int i = 0;

   ProcessListHead=ProcessListPtr=(PLIST_ENTRY)(((char *)PsInitialSystemProcess)+ListEntryOffset);
   if(ProcessListHead != NULL && ProcessListPtr != NULL &&
      ListEntryOffset != -1 && NameOffset != -1)
   {
      while (ProcessListPtr->Flink!=ProcessListHead)
      {
         char ProcessName[16];
         UINT32 processID = 0;
         UINT32 pProcessID = 0;

         Peb=(void *)(((char *)ProcessListPtr)-ListEntryOffset);
         if(Peb)
         {
            memset(ProcessName, 0, sizeof(ProcessName));
            memcpy(ProcessName, ((char *)Peb)+NameOffset, 16);

            processID = (*((UINT32 *)(((char *)Peb) + PIDOffset)));
            pProcessID = (*((UINT32 *)(((char *)Peb) + PPIDOffset)));
            DbgPrint("Name: %s PID:%lu PPID:%lu PEPROCESS: 0x%8x", ProcessName, processID, pProcessID, pEProcess);
         }
         ProcessListPtr=ProcessListPtr->Flink;
      }
   }
   else
   {
      DbgPrint("Probably the ListEntryOffset is wrong!\n");
   }
}
```

Process hiding methods like API interceptions can be revealed by this approach. This technique of tracking objects with the help of list maintained by the kernel will frequently fail to detect objects that are already terminated or which have been hidden by Direct Kernel Object Manipulation (DKOM) techniques.

Is there any solution to detect processed hidden by DKOM techniques?

Lets try other approaches to find a possible solutions.

Method 2:

DKOM is a technique to manipulates the kernel objects to defeat process enumerators. One of the method DKOM uses is to unlink the process from PsActiveProcess list. To detect such hidden processes, we have already discussed few methods like Process BF or iterating through the Handles List. During Process BF, we tried to open a Process using a call "OpenProcess". As we know, OpenProcess opens a process even if it is unlinked from PsActiveProcess. Does that suggest that, OpenProcess uses another list to open a process? Seems so...

If we closely study the NtOpenProcess API (assembly code), we could see that a call to PsLookupProcessByProcessId API to obtain EPROCESS of given pid. PsLookupProcessByProcessId uses PspCidTable to verify the existance of the process. Scanning PsLookupProcessByProcessId API reveals the presence of a list of processes maintained by OS. Lets look at the first few lines of the disassembly of PsLookupProcessByProcessId:

```
PsLookupProcessByProcessId:
   mov edi, edi
   push ebp
   mov ebp, esp
   push ebx
   push esi
   mov eax, large fs:124h
   push [ebp+arg_4]
   mov esi, eax
   dec dword ptr [esi+0D4h]
   push PspCidTable
   call ExMapHandleToPointer
   ...
   ...
```

We can clearly see that ExMapHandleToPointer queries the PspCidTable for the process ID. PspCidTable is a pointer to a HANDLE_TABLE_ENTRY structure.
Structure HANDLE_TABLE_ENTRY can be found by digging through the ntoskrnl.pdb. I have used PDBdump to recover the structure.

```
typedef struct _HANDLE_TABLE_ENTRY
{
    union
    {
        PVOID                   Object;
        ULONG                   ObAttributes;
        PHANDLE_TABLE_ENTRY_INFO InfoTable;
        ULONG                   Value;
    };

    union
   {
       union
       {
          ACCESS_MASK GrantedAccess;
          struct
          {
              USHORT GrantedAccessIndex;
              USHORT CreatorBackTraceIndex;
          };
       };

       LONG NextFreeTableEntry;
   };
} HANDLE_TABLE_ENTRY, *PHANDLE_TABLE_ENTRY;
```

We can iterate through this list to enumerate processes. Understanding the structure of this table may reveal much more internals. This table is used by many non-exported APIs like -

1. **ExCreateHandleTable**
   creates non-process handle tables. The objects within all handle tables except the PspCidTable are pointers to object headers and not the address of the objects themselves.
2. **ExDupHandleTable**
   is called when spawning a process.
3. **ExSweepHandleTable**
   is used for process rundown.
4. **ExDestroyHandleTable**
   is called when a process is exiting.
5. **ExCreateHandle**
   creates new handle table entries.
6. **ExChangeHandle**
   is used to change the access mask on a handle.
7. **ExDestroyHandle**
   implements the functionality of CloseHandle.
8. **ExMapHandleToPointer**
   returns the address of the object corresponding to the handle.
9. **ExReferenceHandleDebugIn**
   tracing handles.

We need pointer to PspCidTable in order to use it to enumerate processes. By searching the function PsLookupProcessByProcessId may give us the pointer to PspCidTable. Once we get the pointer, we need to process it to enumerate all objects in table. Analyze each object to identify the process id.

Rootkits can hide processes from PspCidTable by setting EPROCESS values to 0 (zero) by referencing EPROCESS from parsing the PspCidTable.

Method 3:

In the next method we are going to see how we can enumerate processes by iterating handle table list. The method is similar to the method we discussed in the UserMode approaches. We have seen that when we enumerate handles by using ZwQuerySystemInformation, handles of all the pocesses including that of hidden processes are enumerated. This helps to detect the hidden processes. As we know EPROCESS structure points to a doubly linked list called HandleTableList. Refer to magical offset page for the offset to this list inside EPROCESS on different platforms. By iterating through the list of handle, we can build a list of unique processes
running on the system. We need to find a pointer to HandleTableListHead to iterate through the HandleTableList. HandleTableListHead is a global kernel variable. It is located in one of the kernel file section. How do we acquire a pointer to the HandleTableListHead then? We have pointer to doubly linked list called HandleTableList from EPROCESS + offset, don't we? Alright, let me tell you how to do it.
Acquire a pointer to HandleTableList of any process through EPROCESS + magical offset.

```
PHANDLE_TABLE HandleTable = *(PHANDLE_TABLE *)((ULONG)PsGetCurrentProcess() + HandleTableOffset);
PLIST_ENTRY HandleTableList = (PLIST_ENTRY)((ULONG)HandleTable + HandleTableListOffset);
```

Iterate through linked-list until we find an element that is located in the kernel address space, this element will be HandleTableListHead. To identify Kernel address space, use ZwQuerySystemInformation and enumerate SystemModuleInformation.

```
ULONG Size = 16384;
PSYSTEM_MODULE_INFORMATION_EX Info = NULL;
Info = ExAllocatePool(PagedPool, Size);
NTSTATUS Status;
memset(Info, 0, mSize);
if (Info)
{
   Status = ZwQuerySystemInformation(SystemModuleInformation/*11*/, Info, Size, NULL);

   if(Status == STATUS_SUCCESS)
   {
      ULONG Base = (ULONG)Info->Modules[0].Base;
      ULONG Size = Info->Modules[0].Size;
   }
}
```

Now, iterate through the HandleTableList to acquire HandleTAbleListHead.

```
PLIST_ENTRY Iterator = HandleTableList->Flink;
while (Iterator)
{
   if ((ULONG)Iterator > Base && (ULONG)Iterator < Base + Size) //within kernel address space
   {
      HandleTableListHead = Iterator; // we found the HandleTableListHead
      break;
   }
   Iterator = Iterator->Flink;
}
```

And now, iterate through the HandleTableList using the HandleTableListHead and collect unique processes.

```
PLIST_ENTRY Iterator;
PEPROCESS PeProcess;

Iterator =  HandleTableListHead->Flink;
while(Iterator)
{
   PeProcess = *(PEPROCESS *)((PUCHAR)Iterator - HandleTableListOffset + PEProcessOffset);
   if (PeProcess)
      AddUniqueProcess(QuotaProcess);
}
```

Method 4:

In this method, we are going to enumerate processes from list of threads in the scheduler. Scheduler

maintains list of threads. These lists are doubly linked lists. The list of threads can either be waiting for certain events fired by scheduler or the list of threads ready for execution. These lists are called - KiWaintInListHead, KiWaitOutListHead and KiDispatcherReadyListHead. These lists can be processed to get pointer to ETHREAD structure. Like EPROCESS, ETHREAD structure is made up of several pointers to the process related elements. KiDispatcherReadyListHead list is used by the scheduler when switching address spaces. This list should be used to recognize the owner process of this thread.
The most difficult problem is to find KiDispatcherReadyListHead. A problem lies infact that KiDispatcherReadyListHead address is not present in any of exported functions. Therefore it is necessary to use more complicated search algorithm for its calculation.

After a finding address of thread list, we can easily enumerate their processes by using the following function:

```
void ProcessThreadListHead(PLIST_ENTRY ListHead)
{
   PLIST_ENTRY Iterator;

   if (ListHead)
   {
      Iterator = ListHead->Flink;

      while (Iterator != ListHead)
      {
         CollectUniqueProcess(*(PEPROCESS *)((ULONG)Iterator + WaitProcOffset));
         Iterator = Iterator->Flink;
      }
}
```

Method 5:

The last method, we are going to discuss about is to enumerate processes from intercepting the SwapContext.
The SwapContext function, can be found in ntoskrnl.exe, is called to swap the currently running thread's context with the threads' context that is going to resume its execution. When SwapContext has been called, the value contained in the EDI register is a pointer to the next thread to be swapped in, and the value contained in the ESI register is a pointer to the current thread, which is about the be swapped out. For this interception approach, we need to replace the preamble of SwapContext with a five-byte unconditional jump to our detour function. Detour function should verify that the KTHREAD of the thread to be swapped in, EDI register, points to an EPROCESS block that is appropriately linked to the double linked list of EPROCESS. Once we reach to the EPROCESS block, the life becomes easier. We need to use the magical offsets to retrieve process related information.
The most difficult part of this approach here is to extract the address of SwapContext. By scanning KiDispatchInterrupt API, it is easy to find the address of SwapContext. This is accomplished by scanning some well known memory locations for a signature which consists of the first 20 bytes of the function. If the signature is not found it will scan the whole address space of the ntoskrnl module. And the most critical part of this approach is to UnHook the SwapContext API when done.

What ProcL is *NOT*?
----

1. ProcL can not detect hidden - modules, threads, drivers, files, folders, and registry keys
2. ProcL does not restore any hooks
3. ProcL is not going to keep you Rootkit free!

Support
----

1. ProcL is tested on WinXP.
2. It should run on Win2K.
3. We are testing ProcL on other Windows versions.
4. Next version of ProcL should support Vista.

Conclusion and Future work

Most of the methods discussed in this paper have focused on detecting hidden processes. No detection algorithm is complete or foolproof. As they say, "the art of detection is just that- an art".

As you read this, advanced rootkit and rootkit detection techniques may be developed. Who knows!

We, at Scanit R&D Labs, are continuing our research into detecting hidden processes. As the attacker advances, we will come out with new detection methods.