



RootKit Hunting

Cloaking technology discussed: They can run, but cannot hide...

Authored by: Rajdeep Chakraborty

Email: rajdeep.chakraborty@gmail.com

Wikipedia Description:

A RootKit is a program (or collection of several programs) designed to take fundamental control of a computer system, without authorization by the system's owners and legitimate managers. Typically it works on cloaking technology and a Rootkits act to obscure its presence on the system by masquerading its presence from the operating systems security mechanisms. Often, they carry Trojans as well, thus fooling users into believing they are safe to run on their systems. Techniques used to accomplish this can include concealing running processes from monitoring programs, or hiding files, processes or system data from the operating system.

Rootkits were so named because they allowed an intruder to become a root user (ie, the system administrator) of a Unix system. Since then, similar software has been developed for other operating systems, and the term rootkit has been broadened to include any software that surreptitiously alters an operating system so that an unauthorized user can take arbitrary control of the system.

Rootkits became much better known in 2005, when Sony BMG caused a scandal by including RootKit software on music CDs which altered the Windows OS to allow access to anyone aware of the Rootkits installation. Supposedly, this was done to enforce copy protection of the music on the CDs.

To hunt down Rootkits we have to focus on the concept Rootkits use to cloak or masquerade themselves. However we will look at these from the point of view of a Windows OS (x86 or x64)

What is this masquerade?

As mentioned before, Rootkits take advantage of cloaking technology. In other words, the system information that a user gets with the help of certain applications (tools or utilities) are nothing but an image of the actual kernel image that the kernel passes on to the APIs. It is then that these applications (tools or utilities) display to the user the required information. To explain it in a different way lets assume a 3 tier architecture where the kernel sits at the bottom and at the top level are the applications (tools). The applications in most of the cases do not interact with the kernel level functions themselves. Rather, they take the help of APIs. So we can assume that the APIs become the missing link in the communication process between the applications and the kernel. So these APIs become the middle tier. Now say when we want to see the list of running processes in the system, what usually happens is, the application that's lets us see the running processes makes the an API call (the specific API that aids this activity) and the API in turn communicates with the kernel which then responds with a system call to the actual module that would list the running processes. Once the kernel receives the output, it passes it on to the API which has requested for it and accordingly the information is displayed to the user.

Now the problem is, what if we are not told what the kernel wants us to know? This is what makes the Rootkits technically much more advanced as compared to the other Malwares.

Before I carry on describing this in details, let me make one point very clear... Rootkits are themselves not always malicious or illegitimate. RootKit is a technology that has been used in good used even by some reputed Organizations. However, being a double edge sword, the potential of this technology being used for evil activities are more than using them for legitimate purpose. Malware authors use this technology to masquerade their bad stuff so that they can hide them and elude detection. So to repeat again, Rootkits are not Malware themselves, they are the methodology or technology for eluding or hiding stuff.

So, to carry forward our article, let us look at a scenario where there is a piece of code that injects itself between the bottom layer (in our case it's the kernel) and the middle layer (the APIs). The activities that it would perform is, whenever the API would request something to the kernel, the request would end up going to this piece of code and it would then pass it on to the kernel. Once the requested activity is complete, the kernel will again return back the output to the API, but since this piece of code is sitting in between the API layer and the Kernel, the response of the kernel will be intercepted by the piece of code. Once this response of the kernel is intercepted, then this piece of code filters it according to its needs and returns back the filtered output to the specific API that has initiated the request. This activity of forwarding only the filtered result to the APIs is RootKit activity.

Now to focus in a real life issue, let me take the example of the Peacomm worm. This worm takes the help of RootKit activity to obscure its presence on the system from the users. The Malware binary injects a driver file (.sys) that loads itself in the kernel intercepting calls to modules like **ZwEnumerateValueKey**, **ZwQueryDirectoryFile** etc. It does so by hooking the system functions inside ntdll.dll which in turn allows it to hide files and registry keys that begin with a certain pattern (eg. any file with name noskrnl). In other words, since the worm drops a binary called noskrnl.exe which runs as a process, whenever there is a process listing or whenever we want to see file listings, this RootKit module will filter out the names of any file or process that starts with noskrnl. This would ensure that, to the user or the administrator, this process that is actually running, will never get displayed.

RootKit Detection and Removal:

Detecting Rootkits in this situation is really tough but there are several different techniques that can potentially be used. However new ones are also being developed. None however, are perfect. To make matters worse, RootKit developers are aware of these techniques and are constantly developing their products to evade new detection methods. They mark the RootKit revealer processes and the moment they detect that an AntiRootkit application has been turned on which is in their list, they drop their shield and unhook themselves. The reason being, RootKit revelation applications work basically on one principle. They take an image of the actual kernel data structure and compare it with the image they derive through those APIs. The moment there is an anomaly, they can figure out that some patch up has happened in the kernel modules which is causing this disparity and the process or entry not showing up in the API generated image is actually hiding its presence. Hence, to avoid getting detected by this method, the Rootkits themselves have escalated their level of intelligence, thus dropping guard whenever there is an AntiRootkit application running in the system, then making themselves visible again.

There are over a dozen RKDs (**RootKit Detection System**) available but most are difficult to use or are targeted to detecting specific Rootkits. The names of some of the trusted RootKit revealers are:

- ✍ **Backlight from F-Secure**
- ✍ **RootkitRevealer from SysInternals**
- ✍ **Malicious Software Removal Tool from Microsoft**
- ✍ **RootKit Hook Analyzer**
- ✍ **IceSword**

To find Rootkits manually by following the methods that AntiRootkit applications themselves use, we will take a look at some basic steps. We will be using Windows Debugger or WinDBG. To get access to WinDBG we will install Debugging Tools for Windows. This package can be downloaded freely from Microsoft Website.

Once we open the WinDBG console, we will select the option to debug the local system kernel. This will open the Command window of the debugger. Now we need to download and install the Symbols package for the flavor of Windows OS we are using. Instead of downloading the Symbols package, we can point the Symbols Server component of the WinDBG to the Microsoft Symbols Server. This method is better as it would download the Symbols (.PDB files) for the specific DLL, SYS or EXE as it requires them. To initiate the download of the Symbols required we can use the command **".reload /f"**. Now once the symbols for the Kernel are loaded, we will do the below mentioned things:

1: List running processes and compare with Process Explorer: To list the processes running actually, we will use the command **"!process 0 0"**. This will show us the list of the processes actually running in the kernels data structure. The output looks like:

```
lkd> !process 0 0

**** NT ACTIVE PROCESS DUMP ****
PROCESS 812917f8 SessionId: none Cid: 0004 Peb: 00000000 ParentCid: 0000
  DirBase: 00039000 ObjectTable: e1000b90 HandleCount: 233.
  Image: System

PROCESS ffb56da0 SessionId: none Cid: 0140 Peb: 7ffdf000 ParentCid: 0004
  DirBase: 05db1000 ObjectTable: e15b77d0 HandleCount: 18.
  Image: smss.exe

PROCESS ffb9c020 SessionId: 0 Cid: 01cc Peb: 7ffde000 ParentCid: 0140
  DirBase: 068aa000 ObjectTable: e1496fb8 HandleCount: 234.
  Image: csrss.exe

PROCESS ffb50020 SessionId: 0 Cid: 01e4 Peb: 7ffdb000 ParentCid: 0140
  DirBase: 06aaf000 ObjectTable: e165c1f0 HandleCount: 536.
  Image: winlogon.exe

PROCESS ffb5a980 SessionId: 0 Cid: 022c Peb: 7ffdb000 ParentCid: 01e4
  DirBase: 06e46000 ObjectTable: 00000000 HandleCount: 0.
  Image: services.exe
```

This list will go on till it shows all the running processes.

Once we get the list from the kernel image, we will compare it with the list we are seeing with Process Explorer. An anomaly means the process that is not showing up in the Process Explorer but showing up in the process list of the Kernel image is getting hidden. This is definite RootKit activity.

2: List loaded drivers and compare with Process Explorer: We will do the same for the list of the loaded drivers and compare with the list that's we can see with Process Explorer. The command for this would be "**!mkv**". The output looks like:

```
lkd> !mkv
```

```
start  end      module name
804d7000 806eb780 nt      (pdb symbols)
E:\DebugSymbols\ntoskrnl.pdb\8592B6763F34476B9BB560395A383F962\ntoskrnl.pdb
  Loaded symbol image file: ntoskrnl.exe
  Image path: ntoskrnl.exe
  Image name: ntoskrnl.exe
  Timestamp:      Wed Aug 04 11:49:48 2004 (41108004)
  CheckSum:       002160C9
  ImageSize:      00214780
  File version:   5.1.2600.2180
  Product version: 5.1.2600.2180
  File flags:     0 (Mask 3F)
  File OS:        40004 NT Win32
  File type:      1.0 App
  File date:      00000000.00000000
  Translations:   0409.04b0
  CompanyName:    Microsoft Corporation
  ProductName:    Microsoft® Windows® Operating System
  InternalName:   ntoskrnl.exe
  OriginalFilename: ntoskrnl.exe
  ProductVersion: 5.1.2600.2180
  FileVersion:    5.1.2600.2180 (xpsp_sp2_rtm.040803-2158)
  FileDescription: NT Kernel & System
  LegalCopyright: © Microsoft Corporation. All rights reserved.
806ec000 806ffd80 hal      (deferred)
  Image path: halacpi.dll
  Image name: halacpi.dll
  Timestamp:      Wed Aug 04 11:29:04 2004 (41107B28)
  CheckSum:       000181F2
  ImageSize:      00013D80
  Translations:   0000.04b0 0000.04e0 0409.04b0 0409.04e0
bf800000 bf9c0380 win32k   (deferred)
  Image path: \SystemRoot\System32\win32k.sys
  Image name: win32k.sys
  Timestamp:      Wed Aug 04 11:47:30 2004 (41107F7A)
  CheckSum:       001CCC5F
  ImageSize:      001C0380
  Translations:   0000.04b0 0000.04e0 0409.04b0 0409.04e0
```

This list will go on and on till it shows all the drivers that are loaded up in the kernel.

3: Dump the system Service table and interrupt dispatch table (IDT): To show the **IDT** and the **kiservicetable** we use the command "**!idt -a**". Every address ideally should to point to **nt!kiTrap[something]** or **nt![something]**. If there are entries that are not point to these then, it is obvious that some anomaly is there which doesn't belong to the kernels system call functions. The output looks like:

```
lkd> !idt -a
```

Dumping IDT:

```
00:      804dfbff nt!KiTrap00
01:      804dfd7c nt!KiTrap01
02:      Task Selector = 0x0058
03:      804e015b nt!KiTrap03
04:      804e02e0 nt!KiTrap04
05:      804e0441 nt!KiTrap05
06:      804e05bf nt!KiTrap06
07:      804e0c33 nt!KiTrap07
08:      Task Selector = 0x0050
09:      804e1060 nt!KiTrap09
0a:      804e1185 nt!KiTrap0A
0b:      804e12ca nt!KiTrap0B
0c:      804e1530 nt!KiTrap0C
0d:      804e1827 nt!KiTrap0D
0e:      804e1f25 nt!KiTrap0E
0f:      804e225a nt!KiTrap0F
10:      804e237f nt!KiTrap10
```

```

11: 804e24bd nt!KiTrap11
12: 804e225a nt!KiTrapOF
13: 804e262b nt!KiTrap13
14: 804e225a nt!KiTrapOF
15: 804e225a nt!KiTrapOF
16: 804e225a nt!KiTrapOF
17: 804e225a nt!KiTrapOF
18: 804e225a nt!KiTrapOF
19: 804e225a nt!KiTrapOF
1a: 804e225a nt!KiTrapOF
1b: 804e225a nt!KiTrapOF
1c: 804e225a nt!KiTrapOF
1d: 804e225a nt!KiTrapOF
1e: 804e225a nt!KiTrapOF
1f: 804e225a nt!KiTrapOF

```

Again this list will show the mapping of every memory address that is being used by the kernel.

4: Look for kernel hot-patches: To check which are the functions that has been patched we use the command "**!chkimg -d nt**". This will list down the errors that the debugger has detected. Ideally these errors should not be there. Only if there has been some patching or spoofing activity by some third-party process or code. The output looks like:

```
lkd> !chkimg -d nt
```

```

804d90c9-804d90cd 5 bytes - nt!KiXMMIZeroPage+30
[ fa f7 80 0c 02:e9 08 82 c7 00 ]
804d910c - nt!KiXMMIZeroPage+73 (+0x43)
[ fb: 90 ]
804d9112-804d9115 4 bytes - nt!KiXMMIZeroPage+79 (+0x06)
[ 57 ff ff ff:6d c6 c1 00 ]
804d9545-804d954a 6 bytes - nt!ExAcquireResourceSharedLite+10 (+0x433)
[ fa 8b 75 08 33 db:e9 a3 c2 c1 00 cc ]
804d9564 - nt!ExAcquireResourceSharedLite+98 (+0x1f)
[ fb: 90 ]

```

.....some entries were left out.....

```

804ed809-804ed80f 7 bytes - nt!CcGetActiveVacb+5 (+0x4670)
[ fa 8b 45 08 8b 48 48:e9 ee 7f c0 00 cc cc ]
804ef1dc-804ef1e3 8 bytes - nt!CcSetActiveVacb+7 (+0x19d3)
[ fa 8b 45 08 83 78 48 00:e9 70 66 c0 00 cc cc cc ]
804ef1ff-804ef20c 14 bytes - nt!CcSetActiveVacb+a3 (+0x23)
[ 8b 0a 89 48 48 89 58 50:e9 3d 66 c0 00 e9 2c 66 ]
804f0c70-804f0c73 4 bytes - nt!ExAcquireSharedStarveExclusive+7 (+0x1a71)
[ 64 a1 24 01:e9 cf 05 c6 ]
805745a3-805745a7 5 bytes - nt!NtOpenProcess+5
[ 68 60 a5 4e 80:e9 d8 b4 7b 78 ]
161 errors : nt (804d90c9-805745a7)

```

This shows that these memory addresses and the associated functions have been patched. One thing to note here is, the entry:

```
805745a3-805745a7 5 bytes - nt!NtOpenProcess+5
```

Shows that this address has been tampered with and it calls the function nt!NtOpenProcess that exists inside the ntdll.dll file. Usually this kind of behavior is displayed when some Malware or some RootKit tries to protects its own process by hooking the ntdll.dll and calling the function NtOpenProcess function inside it.

5: Damage control: Now when we have noticed that there was a definite RootKit activity on the kernel to hide itself from the users, we can try to go ahead with replacing these changes done by the RootKit. For these kind of Rootkits, to remove the patching, we will use a command "**!chkimg -f nt**". What this "-f" switch does in here is, it takes the actual kernel data structure image on the disk and overwrites the existing image loaded in the memory. This will uncloak all the hidden processes or files that the RootKit has been hiding till now.

So we come to an end of this small article on the Rootkits act to obscure its presence on the system. There is one word of caution though. If you are not confident with tools like WinDBG or the process of Kernel debugging, then please don't attempt to follow these steps. This is an article for education purpose only and I don't insist anyone to follow what has been taught in this article. If you want to go ahead without proper know-how on windows kernel debugging, then you yourself would be responsible for any system crash or data loss. I will not be responsible for any harm that you may inflict on your system.

DO NOT ATTEMPT THESE STEPS WITHOUT PRIOR EXPERIENCE OR KNOWLEDGE OF THE SYSTEMS INTERNALS AND DEBUGGING TECHNIQUES.