

Windows Internals Lesson 1 (Aditya Kapoor):

Professor Kapoor is a Research Architect at McAfee. He currently writes and creates products to defend against threats. The goal of the upcoming lectures is to learn about stealth and persistence by malicious code. We will also learn about essential windows internals and how malware authors protect their IP.

Between 2006 to 2011, rootkit use was at a very high degree of use and anti-malware services had a hard time catching up. While they are not as popular now, they are still present because attackers want to stay hidden:

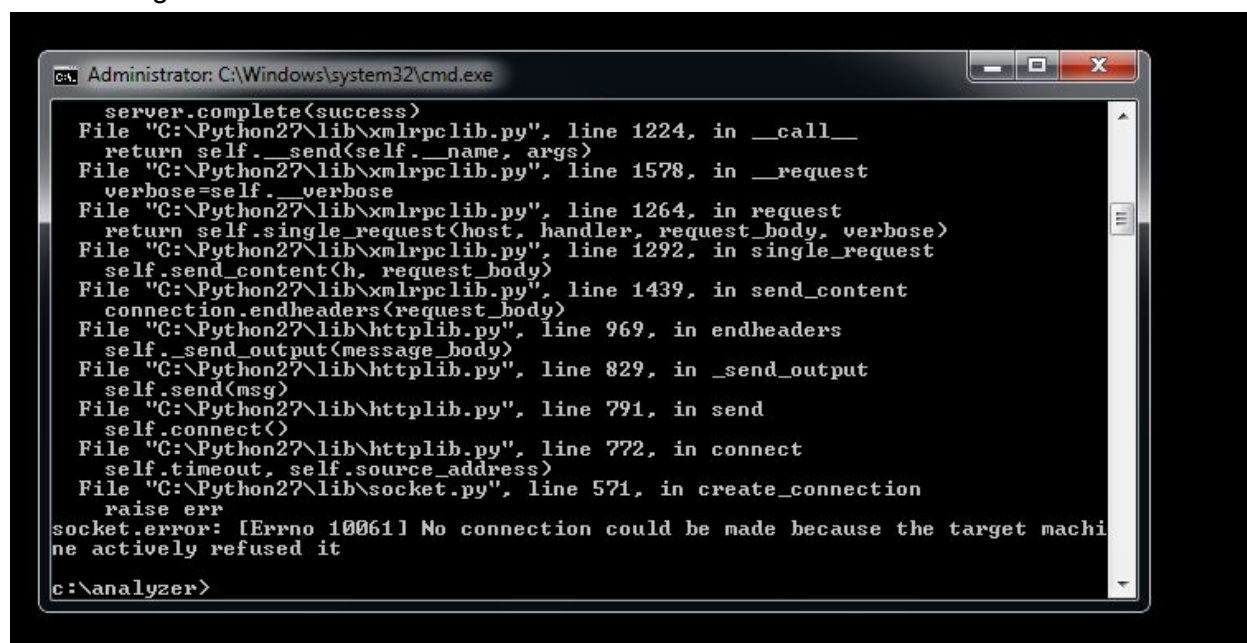
- Approximately 10% of current malware use rootkit
- Rootkits are most prevalent in 32 bit Windows

The focus of this lecture doesn't focus so much on rootkits as it will the vulnerabilities that rootkits exploit:

- Rootkits can infiltrate 64 bit OS Kernel by
 - a) Bypassing driver signing check (e.g. using testsigning mode)
 - b) Modifying the windows boot path (MBR etc) - Secure boot prevents this.
 - c) Kernel exploits in Windows kernel or third party drivers.
 - d) Stealing valid digisigs (Similar to Stuxnet)

So what is the kernel memory? It is a flat memory model. Any kernel driver can access any part of memory (this is likely why they are so dangerous). Big targets for attackers include SSDT, IRP, and IDT.

Next we'll get into the Agony lab. When I used cuckoo to analyze the file in question I ran into the following error:

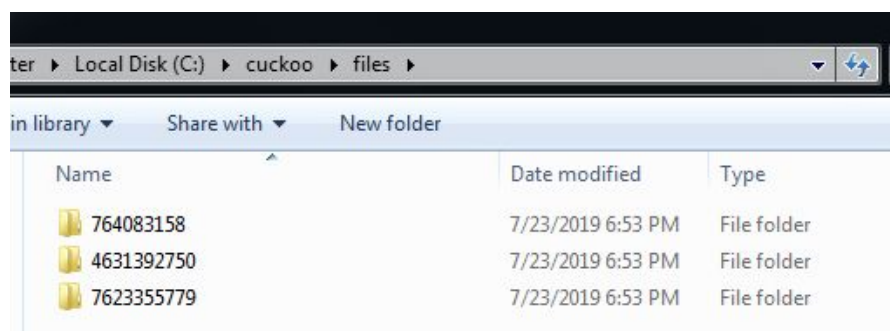


```
Administrator: C:\Windows\system32\cmd.exe

server.complete(success)
File "C:\Python27\lib\xmlrpclib.py", line 1224, in __call__
    return self.__send(self.__name, args)
File "C:\Python27\lib\xmlrpclib.py", line 1578, in __request
    verbose=self.__verbose
File "C:\Python27\lib\xmlrpclib.py", line 1264, in request
    return self.single_request(host, handler, request_body, verbose)
File "C:\Python27\lib\xmlrpclib.py", line 1292, in single_request
    self.send_content(h, request_body)
File "C:\Python27\lib\xmlrpclib.py", line 1439, in send_content
    connection.endheaders(request_body)
File "C:\Python27\lib\httplib.py", line 969, in endheaders
    self._send_output(message_body)
File "C:\Python27\lib\httplib.py", line 829, in _send_output
    self.send(msg)
File "C:\Python27\lib\httplib.py", line 791, in send
    self.connect()
File "C:\Python27\lib\httplib.py", line 772, in connect
    self.timeout, self.source_address)
File "C:\Python27\lib\socket.py", line 571, in create_connection
    raise err
socket.error: [Errno 10061] No connection could be made because the target machine actively refused it

c:\analyzer>
```

As the lecture suggested would happen, a number of folders were created in the cuckoo/files location:



From top to bottom, these folders contain files bad.bin, tzres.dll.bin, and sortdefault.nls.bin.

Tuluka is a third-party tool which can help us to find rootkits. Running it at this point we can sort by 'suspicious' to find the following files:

The screenshot shows the Tuluka 1.0.394.77 by Libertad application. The 'SST' (System Service Table) tab is selected, displaying a list of system functions. The table has columns for 'Index', 'Function', 'Original', 'Current', and 'Reference to'. The first three rows are highlighted in red, indicating they are the functions of interest.

	Index	Function	Original	Current	Reference to
1	119	NtEnumerateValueKey	82e4da0f	a0633480	C:\analyzer\wininit.sys
2	223	NtQueryDirectoryFile	82e61250	a0633050	C:\analyzer\wininit.sys
3	261	NtQuerySystemInformation	82e02775	a0632f00	C:\analyzer\wininit.sys
4	0	NtAcceptConnectPort	82e43a7a	82e43a7a	C:\Windows\system32\ntoskrnl.exe
5	1	NtAccessCheck	82c93f84	82c93f84	C:\Windows\system32\ntoskrnl.exe
6	2	NtAccessCheckAndAuditAlarm	82e7be50	82e7be50	C:\Windows\system32\ntoskrnl.exe
7	3	NtAccessCheckByType	82cbfb2e	82cbfb2e	C:\Windows\system32\ntoskrnl.exe

Note that this image is from the lecture, I could not find these in my VM.

Tuluka also has the ability to restore a program, but it is wise to make sure no other thread is calling a function at the same time (this is a bit unclear to me, but Professor Kapoor said we won't go into it further, so perhaps it isn't super important).

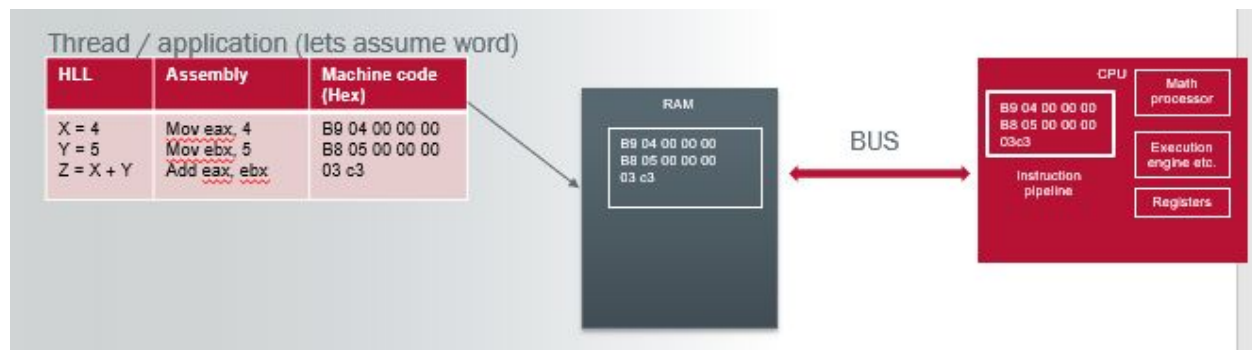
'Hooked' means -- imagine we need to go from point A to B, and something in the middle is filtering packets that meet certain criteria. This "bad guy in the middle" doesn't let the selected packets pass.

The rootkit we analyzed has swapped the old pointer with the new pointer and is filtering files it wants to hide. To better learn about the rootkit, we can activate livekd.exe. Within the kd prompt we can:

- use u to unassemble at an address and see the code at NtEnumerateValueKey
- continually print out code
- .cls clears screen
- !m lists all modules loaded in the kernel memory and their addresses
- !dps can list all the system APIs

The next section covers thread basics. A thread is the smallest unit of execution within an operating system. A machine needs to convert high level language into assembly. Next it is

changed to machine code and loaded into RAM. A bus then takes to instructions to the CPU:



Because a CPU is expected to execute an enormous amount of threads in a short amount of time, an OS uses a thread scheduler.

- One feature of a thread scheduler is to limit how long a given thread is allowed to occupy the CPU.
- Another responsibility is to check which threads have a higher priority.
- Thread scheduler can also boost the priority of threads that have been waiting.

Hyperthreading is utilizing multiple instruction pipelines. In the lecture example one pipeline was reserved for threads that required the math engine. Multiple cores in a CPU was the next step in dealing with multiple threads. Obviously having multiple CPU components helps with the execution of multiple threads (multiple registers, ALUs, execution engines) because the threads don't need to share as much. Remember parallelism.

Thread components are listed on this slide:

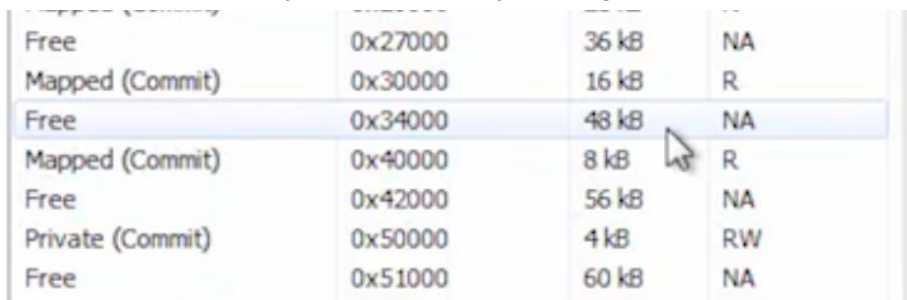
- There are few important concepts of a thread
 - Thread Context
 - Thread Stack
 - Thread Environment Block (TEB)
 - Thread Scheduling
 - Thread - Process relationship
- Thread Object defines a thread
 - Kernel Object are data structure defined by OS to describe various OS constructs, thread being one such construct.
 - TEB, ThreadStack, Context, Priority, State etc are all defined within this struct.

These were not described in detail, so if they come up again in lecture I'll revisit this section of the writeup.

Something I forgot from my Parallel Programming elective is that each thread has its own stack for function calls and local variables.

Processes are implemented as objects, and processes can contain multiple threads (multithreading).

The next lab will help us to understand what process memory looks like. Launching Process Hacker, this tool can lay out the memory starting at address 0x0.

A screenshot of the Process Hacker application showing a memory layout table. The table has four columns: Memory Type, Address, Size, and Protection. The rows are: Free (0x27000, 36 kB, NA), Mapped (Commit) (0x30000, 16 kB, R), Free (0x34000, 48 kB, NA), Mapped (Commit) (0x40000, 8 kB, R), Free (0x42000, 56 kB, NA), Private (Commit) (0x50000, 4 kB, RW), and Free (0x51000, 60 kB, NA). A mouse cursor is pointing at the 8 kB row.

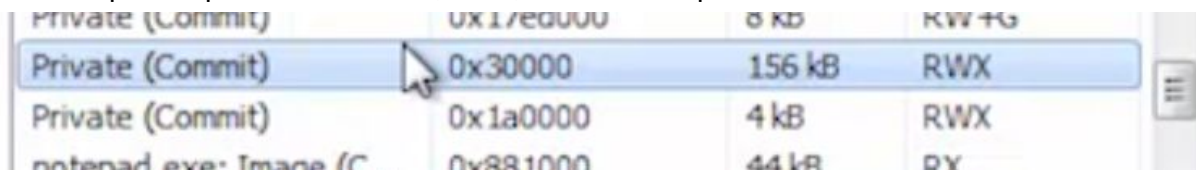
Free	0x27000	36 kB	NA
Mapped (Commit)	0x30000	16 kB	R
Free	0x34000	48 kB	NA
Mapped (Commit)	0x40000	8 kB	R
Free	0x42000	56 kB	NA
Private (Commit)	0x50000	4 kB	RW
Free	0x51000	60 kB	NA

Because we are in a 32 bit system, every process can allocate up to 4GB. Every process has a “Image (Commit)” section that contains the core elements of the process. We can identify which code section is likely to contain executable code by looking for the RX (read-execute) under the Protections header.

Can you enumerate all the processes that are being loaded? Can you list down all of them and show the memory?

Rootkits manipulate applications by modifying the bytes and bits in process memory. They are able to alter the code that executes when a process is run.

Next Professor Kapoor launched malware from the zbot folder. After notepad was closed and launched again, we tried to identify any new processes. The first thing he highlighted was a number of private process with read, write, and execute permissions that had no name listed:

A screenshot of the Process Hacker application showing a list of processes. The table has four columns: Name, Address, Size, and Protection. The rows are: Private (Commit) (0x17e000, 8 kB, RWX), Private (Commit) (0x30000, 156 kB, RWX), Private (Commit) (0x1a0000, 4 kB, RWX), and notepad.exe: Image (C... (0x881000, 44 kB, RWX). A mouse cursor is pointing at the 156 kB row.

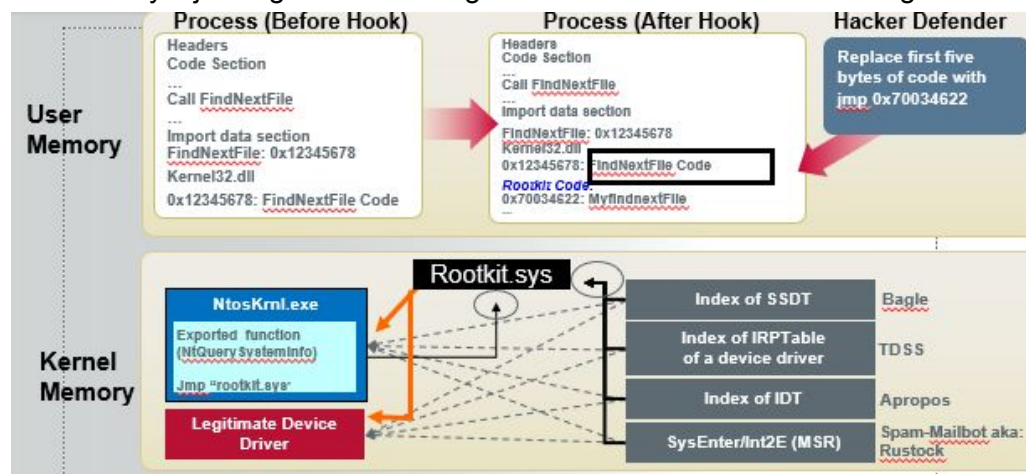
Private (Commit)	0x17e000	8 kB	RWX
Private (Commit)	0x30000	156 kB	RWX
Private (Commit)	0x1a0000	4 kB	RWX
notepad.exe: Image (C...	0x881000	44 kB	RW

Our VM system is now the victim of Process Injection. As notepad was launched, the malware injected its own code into the process.

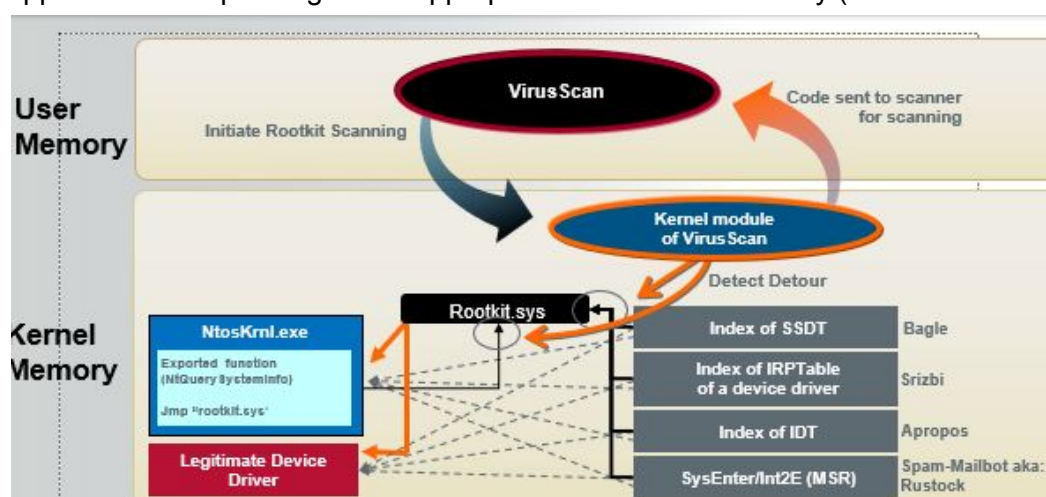
Almost every process running on the machine uses Kernel32.dll.

In the process below, the application calls FindNextFile before the Hook. After the Hook, the address is translated, Kernel32 is found and the rootkit implements the file by replacing the early bytes of the code with a jmp command. Usually a 5 byte op code is the first instruction of

FindNextFile. There are many locations within the kernel that are open for rootkits to modify. It does this by hijacking the calls to legitimate device drivers and filtering the results through it.



How else can we detect this? Antivirus software needs to check and determine if trusted applications are pointing to the appropriate locations in memory (remember the calls to u offset):



Note that although this is a reactive method, McAfee can close vulnerabilities for unaffected users once the threat is identified.

The following process is a Kernel Debugging lab.

It should be noted that even after following the instructions in the supplemental video, I could not get a connection out of WinDbg after setting a breakpoint, so I'm going to try my best to follow along with the lecture.

After a bit of setup, we have a breakpoint of NtEnumerateValueKey:

```
kd> u NtEnumerateValueKey
nt!NtEnumerateValueKey:
82e81a0f 6a5c          push     5Ch
82e81a11 68c80bc982    push     offset nt! ?? ::FNODO
82e81a16 e805f8e0ff    call    nt!_SEH_prolog4 (82c9
82e81a1b 33db          xor      ebx,ebx
82e81a1d 895d94        mov     dword ptr [ebp-6Ch],e
82e81a20 6a08          push     8
82e81a22 59            pop      ecx
82e81a23 33c0          xor      eax,eax
kd> bp 82e81a0f
kd> bl
0 e 82e81a0f    0001 (0001) nt!NtEnumerateValueKey

kd> g
Breakpoint 0 hit
nt!NtEnumerateValueKey:
82e81a0f 6a5c          push     5Ch
kd> .reload /f
Connected to Windows 7 7601 x86 compatible target at ('
Loading Kernel Symbols
```

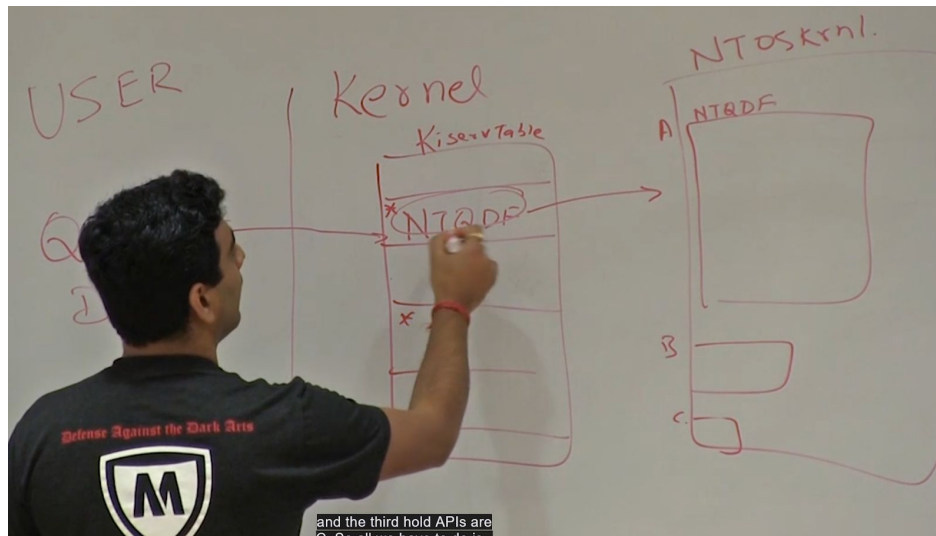
We can view a call stack which is unable to show us any evidence of interception. However we can step through the disassembled code and try to get an idea of what is happening. I think a big thing to keep track of when stepping through the instructions is what addresses are listed or pointed to.

Windows Internals Lesson 2 (Aditya Kapoor):

using the command “dps nt!KiServiceTable L 191” in KDTTool lists the table that has been modified by the rootkit.

When the user makes a query to the directory, a call is made to the Kernel where the KiServiceTable has a series of entries including NTQDF. This is a pointer to a location in

NTOSKernel where the actual NTQDF is located:

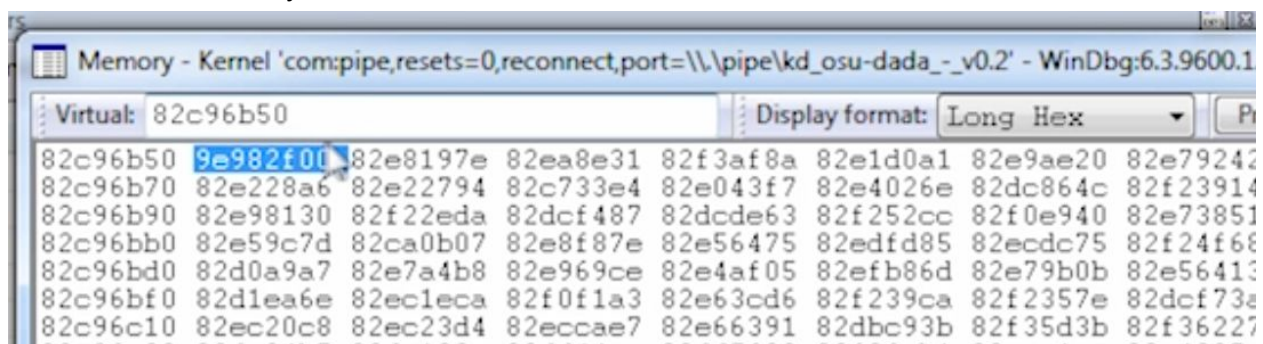


The rootkit we're dealing with replaced the pointer with a pointer to malware. The next step in the lecture process is to repair the pointers in the KIServTable with the original addresses. This will not remove the malware present in memory, but if there is no pointer to it, it cannot be executed.

Stepping through WinDbg, we find the newly added command:

```
82c96b4c 82f34c83 nt!NtQuerySystemEnvironmentValu
82c96b50 9e982f00 wininit+0xf00
82c96b54 82e8117e nt!NtQuerySystemInformationEx
```

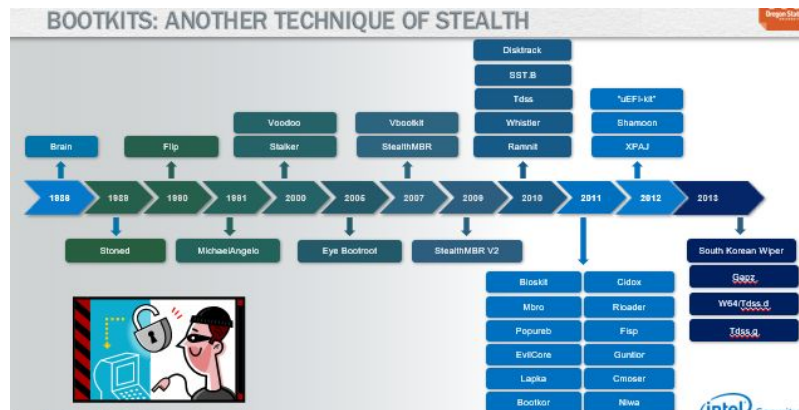
Now enter the memory view, and find the offset:



Within this view, we can manually overwrite the address. Note this is what Teluca does when we right-click and ask the program to fix.

If an attacker modifies the actual function in the NTOSKernal rather than the pointer, that is called an in-line hook.

The first malware was a bootkit called Brain in 1986. I think this is the same instance we heard about in an earlier lecture where Pakistani brothers protected their software. The recent attack on Sony also was done via a rootkit (locking the user out of their system via encryption or a lock, only a ransomed password will recover the files). The timeline is interesting from a historical standpoint but it also highlights just how long rootkits have been used as a method of attack:

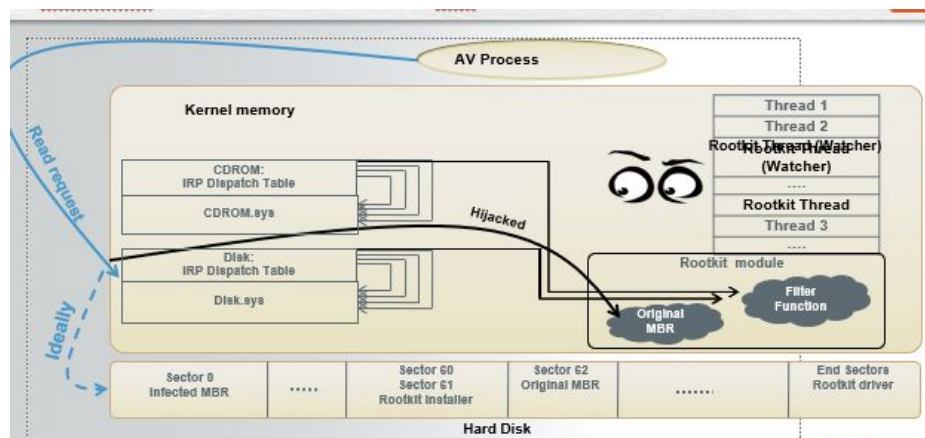


When the PC is turned on, BIOS initializes hardware by calling on code in MBR. MBR loads code from bootsector of the active partition. Bootsector loads & runs bootloader. Once executed, bootloader loads configuration and database from the same partition. If multiple OSs are present, the user is presented with options. Bootloader locates and loads kernel from the disk, then OS has control of the PC.

This series of boots and handoffs allow plenty of opportunities for malware to insert itself into the user's system. It could load before the OS is in control, bypassing many defenses. Microsoft's Secure Boot tries to verify the process at each step.

A generic MBR has 512 bytes of code ending with 5 AAh and then 4 partition table entries. The difference between a rootkit and bootkit is that a bootkit has no stealth component, it just

modifies the MBR. From sector 0 to 62, there is generally nothing.



When the machine boots, the infected MBR will give control to the rootkit installer, which will load the rootkit into memory and then load the original MBR. The eyes in the diagram above refer to a watcher thread that puts the malicious pointers back in the user or anti-malware tries to repair the path.

To look at the details of the disk in question, enter “dt _device_object”:

```
kd> dt _device_object 84b0c738
ntdll!_DEVICE_OBJECT
+0x000 Type           : 0n3
+0x002 Size           : 0x4f8
+0x004 ReferenceCount : 0n0
+0x008 DriverObject   : 0x84b0bcb0 _DRIVER_OBJECT
+0x00c NextDevice     : (null)
+0x010 AttachedDevice : 0x84b0c378 _DEVICE_OBJECT
+0x014 CurrentIrp     : (null)
+0x018 Timer          : (null)
```

You can also navigate to any sector and read the contents of the disk with Roadkil's Sector Editor. First (prior to running the rootkit) we save the first sector of the disk for comparison later to see if anything changes.

The first sign something is wrong is an error telling us “this is not a driver object”:

```
kd> !devstack \device\harddisk0\dr0
!DevObj !DrvObj !DevExt ObjectName
84b0c378 \Driver\partmgr 84b0c430
> 84b0c738 \Driver\Disk 84b0c7f0 DR0
84747100 846f2c10: is not a driver object
84526d30
!DevNode 8454a888 :
DeviceInst is "SCSI\Disk&Ven_VMware_&Prod_VMware_Virtual_S\4&3b501
ServiceName is "disk"
```

In the lecture, something went unexpectedly for the instructor and the demo was abandoned. We would have seen some modification to the driver object of the disk.

Some of the trends in the stealth malware world include the following:

- File Forging is where rootkits modify a file so that when you attempt to read the file you are presented with a different view
 - this is better than hiding files because hidden files can be located with file-system parsers
- Memory Forging is using a rootkit for self protection, attacking AVs (what are AVs?), and “some other things”

Some self defense methods include identifying AV, untrusting the AV, and whitelisting programs you know to be safe. Nothing else is allowed to run on the system. Naturally this is a bit restrictive, so it's very important to detect and block rootkit systems.

- Behavioral identification of AV
 - Identify the AV activity by monitoring the behavior of the process or thread, if it triggers their behavioral detection logic, the AV is terminated.
- Untrusting the AV and whitelisting of legitimate applications
 - Trust based deterrence in the rootkits
 - Threats establish trust on the essential drivers for the system and everything else could be locked out.
 - AV now has to find ways to get trusted by malware to get a chance to even load.

Testing is expensive, because it has to be in a dynamic environment. Many OS versions have to be checked.

Good question was asked: Our process broke into the machine that was being patched with WinDbg. How would an anti-malware software do this? The software HAS to halt all other processes when sending the patch to the kernel.

Next we're looking at a Brazilian banker Rootkit. This rootkit is particularly dangerous because it has the ability to bypass driver signing requirements and prevent security tools from loading.

Once it registers for a callback, it modifies in memory any process that may detect it.

```
cmp     eax, esi
je      plusdriver+0x53e4 (872053e4)
mov     dword ptr [eax], 1B8h
mov     dword ptr [eax+4], 8C2C0h
push    edi
```

Why use “mov eip eip” instead of no op? Instructor was not sure.

The following was presented as “more bits and pieces of information”:

- Uses bcdedit to bypass driver signing requirements
 - DISABLE_INTEGRITY_CHECKS
 - TESTSIGNING ON
- Prevents security tools from loading
 - OS callbacks such as SetLoadImageNotifyRoutine
 - Also used by BlackEnergy rootkit
 - Image identification by name
 - Generic security tool identification

Taking a closer look, this information is more detail on the methods used by Brazilian Banker Rootkits.

Next demo involves first checking if this particular function is hooking NtEnumerateValuekey or not. After confirming this in memory, check for a byte sequence. We can write it with WRITEMEM in the debugger. We are using bytes because while strings can change, functionality is less likely to change.

Another demo we didn't seem to get to involved finding out the code responsible for modifying pointers. This is similar to the Agony process, but we didn't interrupt the rootkit modifying the SSDT table. We need to find the offset in the table where the target API is located, create a breakpoint (BAW1), and write a byte. I think there's an intermediate step, but the finale is that we are led to the portion of the rootkit that does the modifications. Debugger also has a feature to filter loaded modules. This is helpful to see what each driver is doing.

