PREDICTING THE FUTURE OF STEALTH ATTACKS

Aditya Kapoor, Rachit Mathur
McAfee Inc., 20460 NW Von Neumann Dr.,
Beaverton, OR - 97006, USA

Email {Aditya_Kapoor, Rachit_Mathur}@
McAfee.com

ABSTRACT

This paper takes an in-depth look into the attack strategies of recent rootkits and analyses what has worked for them. In doing so it highlights some of the profitable attack methodologies from the perspective of kernel rootkits. The discussion in this paper about prediction of the future of stealth attacks is derived from our analysis of multiple rootkits over many years and also based on current trends and some specific techniques. The main aim of this discussion is to help reanalyse rootkit defences and decide what technological improvements (if any) are needed in current and future products to better combat the ever changing stealth threat landscape.

ROOTKITS, STILL A PROBLEM?

It takes time for an attack vector to become mainstream and even more time to be used on a large scale. Rootkits started showing up in the wild in the mainstream in 2004–2005 and were constantly adopted by malware authors. In the same time period most AV products were working on overhauling their defences. So assuming everyone can detect rootkits, if a new one comes out, one can just write a new 'signature' for it and close the case? Right? No, not really; the problem is that a new rootkit with a new technique has the ability to force AV/anti-rootkit tools to update their core technology of detection and remediation over and again, unlike any other malware class. So it is a good idea to have a reality check once in a while to see the current technology changes as well as quantify the problem.

We will look at the rootkit technology changes in later sections; for now let's just look at the statistics for quantification purposes. Statistics around malware are usually a moving target and dependent on the individuals who are doing the research. Another problem in collecting reliable statistics in the case of rootkits (as well as malware in general) is that it is hard to account for what you don't know. So we cannot really calculate the number of rootkits we don't know about. However, when we queried our consumer data for a month (for the rootkits we knew about) we found approximately 10% of the detections were rootkit related. We backed this number up by querying the internal database that tracks the total number of submissions to *McAfee* in a year; we then calculated how many of the submissions were rootkit related and the number came out to be close to 8%. In the past few years other vendors such as *Microsoft* [1] and *Symantec* have also published their stats and the numbers were in the range of 7–10%. For the sake of this discussion we believe that the average 8–10% rootkit count is close to realistic as a number that represents the infected machines.

These statistics demonstrate that the rootkit problem is still a relevant one. However, it is comparatively rare that we 'hear' about a rootkit causing problems for our support team or customers. Usually the infection is taken care of by existing technologies for detecting rootkits. Although every once in a while when a cleverly crafted piece of code comes around we 'hear' about it. Once a solution is provided for a new rootkit the next question in our minds is 'what is going to be the next step for this family?' Analysing these trends has usually helped us in understanding the bigger picture and in keeping the technology prepared to reduce our response times.

In the next few sections we look at the trends of the major changes in rootkit techniques over the last decade. We also present case studies of some of the most interesting rootkits that are prevalent in 2011. We discuss what has worked for these rootkits and what inference we can derive from their analysis for future trends.

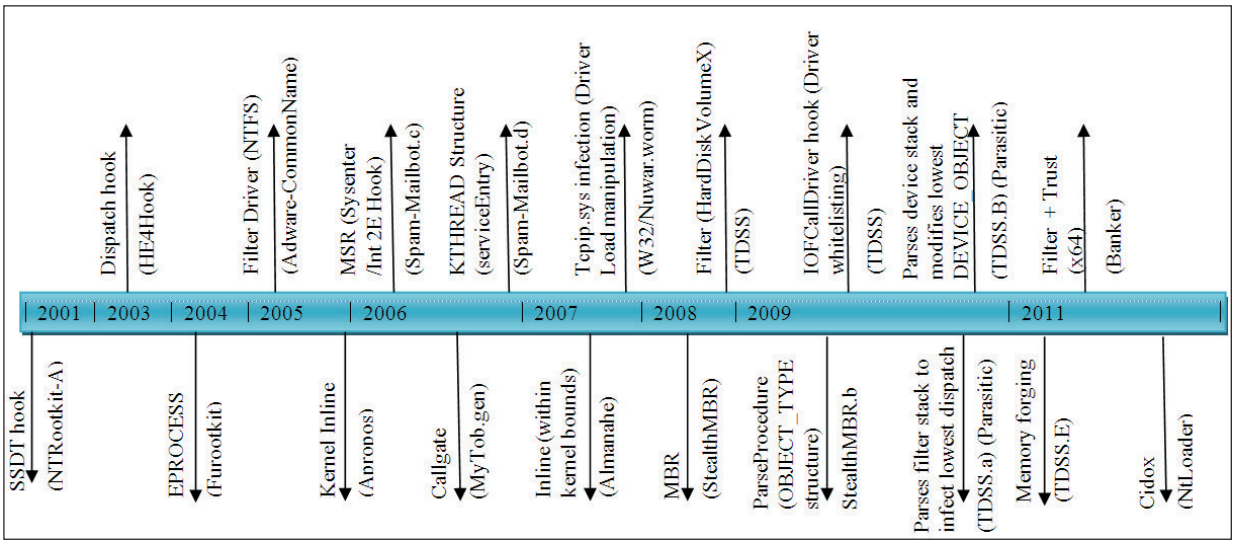

Figure 1: New rootkit techniques introduced over the last decade.

ROOTKIT ATTACKS IN THE LAST DECADE

Many kernel–mode techniques have been released in the last decade which continue to fuel the growth of rootkits. This section presents some of the major changes in the techniques to give a sense of the pace and type of change. Furthermore, taking a holistic view of trends might give us some idea of the stealth techniques that may become popular in the future.

Figure 1 shows the timeline of major new techniques as they were released. The rootkits on this timeline were the first to release the specific techniques that we found in the wild. These techniques were then followed by many more rootkit families.

The concept of manipulating the *Windows* kernel for stealth started with NTRootkit-A which hooked SSDT. HE4Hook [2] extended the idea by hooking the dispatch table of NTFS which became extremely popular among rootkits and is still being used by the likes of TDSS and Cutwail. Fu rootkit [3] came up with an entirely new attack for process hiding in 2004, which became almost a standard. Most of the process hiding rootkits to date use the technique of EPROCESS doubly linked list modification. 2005 saw exponential growth in adware and to protect its files, the CommonName family came up with the simple yet effective NTFS filter driver, which created a lot of problems for AV vendors who themselves use filter drivers for detection and repair.

The rootkit Apropos [4] came out towards the end of 2005 and used a clever combination of inline hooks and interrupt table hooking. To gain control it would create an exception in the kernel code of NtQueryDirectoryFile and the exception was handled by an interrupt routine in IDT that was also hooked by the rootkit. The main challenge here was that the interrupt-driven hook disassociated the kernel inline hook from the memory of the owning module, thus finding the malicious module and removing it became a challenge. We will further discuss memory disassociation challenges later on in this paper.

Following a brief lull in new techniques, the Rustock (spam-mailbot.c) family implemented a rootkit that used SysEnter/int2E hooks in addition to hiding its files using alternate data streams. Another trick that Rustock delivered was to hook SSDT by using KTHREAD [5] object manipulation. This allowed the rootkit to selectively hook SSDT on a per thread basis. This technique is currently being used by the BlackEnergy rootkit. In addition, Rustock also searched the lowest device objects and sent IRPs directly to it to bypass filter drivers.

In the same year (2006), rootkit components of the prevalent Mytob family started using call gates to hide their process by writing to ring 0 using \device\physicalmemory, without the need to install a kernel-mode module.

By this time most of the AV tools had started to include multiple range check and cross diff approaches in their products to find the discrepancies in kernel memory or disk. Almanahe [6] fought back in 2007 by improving the kernel inline technique, such that the hooks would lead the tools within the *Windows* kernel module range and thus fail to raise suspicion.

Nuwar's rootkit released in 2007 made an effort to minimize its footprint by removing dependency on registry keys to load its device driver during reboot. It rather parasitically infected tcpip.sys and wrote its loader code in the file; during machine boot the infected tcpip.sys would load the malicious device driver in its context.

The rootkit authors soon realized that modifying an existing kernel driver would benefit them further in stealth. In 2008 Rustock quickly took advantage of this technique by overwriting less critical drivers like beep.sys [7]. Similarly, the TDSS family of rootkits further improved the Nuwar strategy by parasitically infecting random boot drivers to write their loader code and storing the actual driver in RAW sectors so that there is no file association on disk.

Moving to the right on the timeline, various other rootkit techniques covered the breadth and depth of the kernel. In the last few years rootkits have moved in many directions to evade detection. They have manipulated various kernel objects like 'OBJECT_TYPE', 'DEVICE_OBJECT', 'DRIVER_OBJECT' and 'KTHREAD', in addition to coming up with direct attacks on anti-rootkit tools and evasion tactics. From this, one can observe that the profitable area of the kernel has been a moving target. For a known technique, there can always be some solution to counter it. So, as expected, once a technique is known about and becomes popular, most anti-rootkit tools detect it and the malware needs to find other techniques and places to achieve stealth.

Even though the myriad of techniques present a grim outlook in terms of predicting what will come next, there are certainly some areas which we believe could become more popular than others. We describe them in some of the following sections.

CURRENT STATE OF ROOTKIT ATTACKS

We analysed lots of prevalent rootkits in order to discuss the five rootkits which we found useful for this discussion.

BlackEnergy rootkit

The BlackEnergy trojan has been featured in the media for its powerful botnet [8] which is driven by its desire to steal banking credentials. It has been evolving since 2008 and it uses various stealth techniques to stay hidden for a longer time. Current variants of BlackEnergy use the following two techniques to thwart detection:

1. KThread manipulation

This trojan makes use of the fact that each thread can have its own SSDT. The KTHREAD structure has an entry called ServiceTable which holds the address of the SSDT to use for system service requests by that thread. The malware does not need to hook functions in the SSDT pointed to by NT!KiServiceTable which is monitored by all anti-rootkit tools. Instead the malware replaces the ServiceTable pointed to in the various threads. As described earlier, this technique was first implemented by Rustock in 2006.

2. OS callbacks

Instead of defence, this malware attacks AV by preventing AV processes from loading. It registers notification callbacks using functions such as PsSetLoadImageNotifyRoutine.

When a process is loading, the malware gets a callback by the OS where it first checks if the loading process is on its

blacklist of known AV processes. If it isn't, then the malware allows the process to load normally. But if it is an AV process, the malware writes a return instruction (0xc3) at the entry point in the memory of the process and then allows it to load. The AV process typically terminates immediately due to this patch by the malware. This technique was first reported to be used in the wild by Nuwar back in 2007.

Koutodoor rootkit

The Koutodoor family has been in the wild since at least 2007. Currently this rootkit is most prevalent in China with almost 96% of all detections being reported from China. It uses polymorphic droppers and we continue to receive a lot of samples of this rootkit family.

For its stealth activity Koutodoor uses the following strategies:

3. ParseProcedure hook

It replaces the ParseProcedure function pointer for device objects. IoDeviceObjectType is an exported symbol which points to the OBJECT_TYPE structure in the header of device objects. The OBJECT_TYPE structure contains the OBJECT_TYPE_INITIALIZER structure which in turn has a pointer to the ParseProcedure function. Koutodoor changes the value of this ParseProcedure function to point to its own routine which prevents access to the malware's files on disk [9].

It has a minor difference from some StealthMBR variants that also hijacked ParseProcedure. The difference is that StealthMBR hijacked the OBJECT_TYPE field in the OBJECT_HEADER of specific device objects of \driver\disk. Since the OBJECT_TYPE eventually contains the ParseProcedure function pointer, StealthMBR was also able to have its own function there. Koutodoor directly hijacks ParseProcedure in the OBJECT_TYPE pointed to by IODeviceObjectType which affects all device objects as opposed to specific device objects of a particular driver object. Some anti-rootkit tools were not able to detect one or the other.

4. File/memory disassociation

Another interesting technique of Koutodoor is that it flip-flops its file names on every reboot. On boot, once the rootkit loads into memory, it copies its sys file (on disk) as a different file name and deletes the original one. This new file is protected by the rootkit in memory. It then creates a service registry entry to load the newly created file the next time the system boots, and protects that registry as well. So it will not help even if a memory scanner finds the ParseProcedure hook and locates the module on disk that is associated with the hook memory location, because the file that loaded has been deleted.

Additionally, Koutodoor does not hide its file; instead it denies read/write access to it. Hidden files can sometimes be easier to detect than protected files. This technique (although not exactly a stealth technique) needs to be watched out for by security products more generically, because this will be the end result of stealth.

TDSS rootkit

Currently in its fifth generation, TDSS has been constantly evolving to counter anti-rootkit defences. A recent article by *Kaspersky* [10] highlighted that TDSS is now one of the biggest botnets with 4.5 million machines infected with TDSS. TDSS has been a moving target and has been changing its rootkit techniques quickly enough to remain undetected by most AV vendors.

As is evident on the trends timeline, in 2008 the variants of this rootkit were using filter drivers to hide their files and traffic along with using inline kernel hooks on APIs like NtFlushInstructionCache and NtEnumerateKey for hiding registry keys and gaining control. By the time various AV vendors caught up with the detection of these APIs, the new versions came out that were using yet another API hook (IofCallDriver and IofCompleteRequest) by directly modifying the kernel. The recent variants of TDSS have updated their attack strategies; some of the current ones are listed below:

5. Object hijacking

In late 2010, TDSS began hijacking the driver object of the device attached below the DR0 disk device. The lower device is obtained by following the device object pointed to by the DR0_DeviceObject-> DeviceObjectExtension->AttachedTo field. Let us call this Lower_DeviceObject. These variants replaced the driver object pointed to by the Lower_DeviceObject->DriverObject field.

```
CLEAN:
kd> !devstack \device\harddisk0\dr0
  !DevObj   !DrvObj            !DevExt   ObjectName
  81ba7318  \Driver\PartMgr    81ba73d0
> 81ba7540  \Driver\Disk       81ba75f8  DR0
  81be2030  \Driver\atapi      81be20e8  IdeDevicePOTOLO-3

INFECTED:
kd> !devstack \device\harddisk0\dr0
  !DevObj   !DrvObj            !DevExt   ObjectName
  81ba7318  \Driver\PartMgr    81ba73d0
> 81ba7540  \Driver\Disk       81ba75f8  DR0
Invalid type for DeviceObject 0x81be2030
```

Figure 2: Device stack before and after TDL4 infection.

Figure 2 shows the device stack in *WinDbg* output before and after infection. The first thing that stands out is the error message that the DeviceObject type is invalid. Note that the device object is still the same as on a clean machine (0x81be2030). Figure 3 shows further inspection of this device object. It is evident that the malware has changed the Type field to zero, which is causing the debugger extension to break. But noticeable here is the hijack of the driver object. This device object suddenly starts to claim that it belongs to some other driver object now.

```
CLEAN:
kd> dt _device_object 81be2030
nt!_DEVICE_OBJECT
   +0x000 Type             : 3
   +0x002 Size             : 0x234
   +0x004 ReferenceCount   : 0
   +0x008 DriverObject     : 0x81be4030 _DRIVER_OBJECT

INFECTED:
kd> dt _device_object 0x81be2030
nt!_DEVICE_OBJECT
   +0x000 Type             : 0
   +0x002 Size             : 0x234
   +0x004 ReferenceCount   : 0
   +0x008 DriverObject     : 0x818cf2d0 _DRIVER_OBJECT
```

Figure 3: Device object before and after TDL4 infection.

Figure 4 further inspects the hijacked driver object. The type field of this is also set to zero and it is very interesting to note that the DriverName field of this driver object is also \driver\atapi. This is the fake and malicious driver object.

```
kd> dt 0x818cf2d0 _DRIVER_OBJECT
nt!_DRIVER_OBJECT
   +0x000 Type             : 0
   +0x002 Size             : 168
   +0x004 DeviceObject     : 0x817eda20 _DEVICE_OBJECT
   +0x008 Flags            : 4
   +0x00c DriverStart      : 0xf982d000
   +0x010 DriverSize       : 0x17480
   +0x014 DriverSection    : 0x81bf1d70
   +0x018 DriverExtension  : 0x81be40d8 _DRIVER_EXTENSION
   +0x01c DriverName       : _UNICODE_STRING "\Driver\atapi"
   +0x024 HardwareDatabase : 0x8066e9d8 _UNICODE_STRING "\REG
   +0x028 FastIoDispatch   : (null)
   +0x02c DriverInit       : 0xf98425f7     long  atapi!GsDri
   +0x030 DriverStartIo    : 0xf98347c6     void  atapi!IdePo
   +0x034 DriverUnload     : 0xf983e204     void  atapi!IdePo
   +0x038 MajorFunction    : [28] 0x81474439     long  +fffff
kd> dd 0x818cf2d0+0x38
818cf308  81474439 81474439 81474439 81474439
818cf318  81474439 81474439 81474439 81474439
818cf328  81474439 81474439 81474439 81474439
818cf338  81474439 81474439 81474439 81474439
818cf348  81474439 81474439 81474439 81474439
818cf358  81474439 81474439 81474439 81474439
818cf368  81474439 81474439 81474439 81474439
818cf378  818cf2d0 00000000 00000000 00220020
```

Figure 4: TDL4 malicious driver object.

Figure 5 shows the difference in the output of the drvobj debugger extension before and after infection. There are two things to note here. Firstly, the device object we were inspecting from the stack 0x81be2030 is now missing from the device object list. Secondly, the driver object in both cases is still 0x81be4030.

```
CLEAN:
kd> !drvobj \driver\atapi
Driver object (81be4030) is for:
 \Driver\atapi
Driver Extension List: (id , addr)
(f98418d8 81be51f0)
Device Object list:
81be25b8  81be2030  81be3030  81ba9030

INFECTED:
kd> !drvobj \driver\atapi
Driver object (81be4030) is for:
 \Driver\atapi
Driver Extension List: (id , addr)
(f98418d8 81be51f0)
Device Object list:
81be25b8  81be3030  81ba9030
```

Figure 5: Driver\atapi before and after TDL4 infection.

To sum up, the device object of \driver\atapi which resides on the stack has been removed from the device object list of atapi. But it still resides on the stack. And the driver object field in that same device object has been hijacked to point to a new fake and malicious driver object. The bottom half of Figure 4 shows the dispatch table of this malicious driver object and we can see that all the major functions are pointing to the same location which is the entry point for the malware's filter routine.

Doing this allows the malware to intercept IO flowing through this stack without needing to hook the dispatch table of the driver below the disk device ('driver\atapi' in this example). So scanners that scan the dispatch table by obtaining the driver object of 'driver\atapi' will not see any problems.

6. MBR and raw sector writes

A new variant of TDSS appeared in mid-2010 that started infecting the MBR. It also hides the infected MBR from any program that tries to read it and presents the clean MBR instead. By infecting the MBR, the threat does not require any file or registry to persist across the reboot which is arguably better. It stores its 'files' in empty sectors typically after partition end that are not part of the *Windows* file system and are reconstructed and loaded by the malware during boot.

This technique was first introduced by StealthMBR, which also used to keep its components towards the end of the disk. TDL4 improved on this by creating its own encrypted file system towards the end of the disk instead of leaving things visible [11].

Infecting the MBR provides early load abilities and thus the opportunity for the malware to disable AV tools as well as OS protection features like PatchGuard, debugging support etc. before the protections can be loaded. Raw sector writes have the additional advantage of remaining invisible to traditional file scanners. This technique has been used successfully for quite some time now and will continue to gain popularity among mainstream rootkits.

7. Memory forging

In early 2011, TDSS.E appeared [12]. This is the first rootkit in the wild to forge kernel memory contents. There are some proof-of-concept methods already documented to forge memory contents [13, 14].

The lure to do this for the malware authors is obvious. Many anti-rootkit scanners rely on their view of the memory to be able to detect. If their view of system memory is compromised then they may not be able to detect the simplest of malware hooks. But forging memory contents is not as simple as hooking an OS function. A detector does not have to use any OS routine to read memory. It can directly read memory using instructions like mov.

TDSS.E uses dispatch table hooks to conceal the driver that it parasitically infects. It is then able to fake the memory location where it installs these dispatch table hooks. When a read request comes for that location from a detector the rootkit is able to intercept the read and make the detector read from a faked location instead. This dupes the detector into believing that there are no hooks when, in fact, the dispatch table is actually hooked.

```
_CONTEXT
    +0x000 ContextFlags     : Uint4B
    +0x004 Dr0              : Uint4B
    +0x008 Dr1              : Uint4B
    +0x00c Dr2              : Uint4B
    +0x010 Dr3              : Uint4B
    +0x014 Dr6              : Uint4B
    +0x018 Dr7              : Uint4B
    +0x01c FloatSave        : _FLOATING_SAVE_AREA
    +0x08c SegGs            : Uint4B
    +0x090 SegFs            : Uint4B
    +0x094 SegEs            : Uint4B
    +0x098 SegDs            : Uint4B
    +0x09c Edi              : Uint4B
    +0x0a0 Esi              : Uint4B
    +0x0a4 Ebx              : Uint4B
    +0x0a8 Edx              : Uint4B
    +0x0ac Ecx              : Uint4B
    +0x0b0 Eax              : Uint4B
    +0x0b4 Ebp              : Uint4B
    +0x0b8 Eip              : Uint4B
    +0x0bc SegCs            : Uint4B
```

Figure 6: Some context structure fields.

This rootkit uses the KiDebugRoutine hook technique to fake memory. This technique requires the rootkit to do two things. Firstly, hook the KiDebugRoutine pointer to point to its own malicious routine. And also set a hardware breakpoint (DRX register setting) to monitor access to the memory area of the

kernel (dispatch table) that it patches. It sets the DR0 register to the address of that location and sets DR7 flags to set monitoring for read access. The malware uses the KeQueryActiveProcessors and KeSetSystemAffinityThread functions to execute its thread on each processor of the system and set the hardware breakpoint on each of them.

Now, whenever a read access occurs to this dispatch table location, a hardware breakpoint is triggered and an exception is raised. The system saves the context of the current executing thread which raised an exception and invokes the system's exception processing routines. This eventually causes a call to the function pointed to by KiDebugRoutine and this is how the malware gets control on read access to dispatch hook location. If the exception is handled successfully the system restores the context of the thread that raised the exception and resumes its execution. Figure 6 shows some of the fields in the context structure that is saved for the thread. This will give the reader an idea of the type of information about the thread that gets saved. If some fields are modified during exception handling then the original thread resumes execution with those modified registers and environment.

Note that the function pointed to by KiDebugRoutine will also be called on other exceptions and hardware breakpoint events. In this handler the malware does two things. Firstly, it needs to ascertain that it is being invoked due to the hardware breakpoint it set for the dispatch table hook location. And secondly it needs to check who is requesting access to this location and depending on that either fake memory or allow access. Let us look into how this is achieved.

Figure 7 shows the beginning of the malware's KiDebugRoutine handler code. If it encounters a breakpoint, then the malware simply increments the instruction pointer for the thread where the exception occurred and returns it as handled. Otherwise the jump is taken. External debuggers (like *WinDbg*) stop working as a result of this.

```
sub     esp, 10h
mov     eax, [esp+10h+PEXCEPTION_RECORD]
mov     eax, [eax+EXCEPTION_RECORD.ExceptionCode]
push    ebx
push    ebp
push    esi
xor     edx, edx
push    edi
mov     edi, [esp+20h+PCONTEXT]
or      [edi+CONTEXT.ContextFlags], CONTEXT_DEBUG_REGISTERS
cmp     eax, EXCEPTION_BREAKPOINT
mov     [esp+20h+var_C], edx
mov     [esp+20h+var_8], edx
mov     [esp+20h+var_10], edx
mov     [esp+20h+var_4], edx
jnz     short loc_403BCC ; take the jump if not a break point
inc     [edi+CONTEXT._Eip] ; resume execution at exception EIP plus one

                        ; CODE XREF: KiDebugRoutine_Hook+89↓j
                        ; KiDebugRoutine_Hook+4F4↓j ...
mov     al, 1           ; set as handled
jmp     loc_4040E4      ; done!
```

Figure 7: TDSS.E KiDebugRoutine handler part-1.

Figure 8 shows the code at the target of the jump in Figure 7. This code checks the instruction pointer (EIP) saved within the context of the thread where the exception occurred. So this will be the EIP where the exception was raised. If the exception occurred at a kernel-mode address then the jump is taken.

```
mov     eax, large fs:1Ch
mov     ecx, ds:MmHighestUserAddress
mov     [esp+20h+PCONTEXT], eax
mov     eax, [edi+CONTEXT._Eip]
cmp     eax, [ecx]
jnb     loc_404026      ; jump is taken when it is a kernel mode address
```

Figure 8: TDSS.E KiDebugRoutine handler part-2.

At the target of this jump, firstly the exception is processed normally by checking access flags, clearing the DR6 register, etc. Figure 9 shows the code to identify if this is a read access to the monitored memory and if the malware wants to block this particular access. If both of these conditions are true, then the read location is altered to point to a memory area with contents the malware wants to forge. The malware assumes that a detector will use the ESI register to point to its read location. If ESI in the saved context is found to be pointing to the location of the dispatch hook, the malware modifies the ESI value in the context to now point to another memory location and marks the exceptions as handled. When the original thread resumes and its context is restored, its ESI register points to a different location than before the exception was raised. This new ESI location contains the benign contents the malware wants to forge. And this is how it fakes memory. To summarize, when a detector tries to read, a hardware exception is raised, and the detector's thread goes to sleep. Due to the malware's hook, when the detector's thread wakes up after the exception has been processed, the detector's ESI is no longer pointing to the area intended and the detector ends up reading from another location thinking it is reading the dispatch table.

```
mov     edx, dword_41D814
cmp     [esp+20h+DR0], edx ; is this due to the DR0 being set?
jnz     short loc_4040E2 ; set as not handled and return
mov     eax, [edi+CONTEXT._Eip]
cmp     eax, dword_41D810 ; compare EIP with pre-defined locations
                          ; that are allowed to read correct memory
jz      loc_403BC5        ; set as handled and return
cmp     eax, dword_41D78C
jz      loc_403BC5        ; set as handled and return
cmp     eax, dword_41D634
jz      loc_403BC5        ; set as handled and return
mov     ecx, ds:MmHighestUserAddress
mov     ecx, [ecx]
cmp     eax, ecx
jb      short loc_4040E2 ; set as not handled and return
jbe     short loc_4040E2 ; set as not handled and return
mov     eax, [edi+CONTEXT._Esi]
mov     ecx, dword_41D818
cmp     eax, ecx          ; is ESI pointing to protected memory region?
jz      short loc_4040CF ; jump to fake_memory
cmp     eax, edx
jnz     loc_403BC5        ; set as handled and return
cmp     eax, ecx
jnz     loc_403BC5        ; set as handled and return
loc_4040CF:               ; CODE XREF: KiDebugRoutine_Hook+52E↑j
mov     eax, dword_41D830
add     eax, 38h
mov     [edi+CONTEXT._Esi], eax ; set thread ESI to fake memory location
jmp     loc_403BC5        ; set as handled and return
```

Figure 9: TDSS.E KiDebugRoutine handler part-3.

The implementation is not perfect and detectors can watch out for hardware breakpoints before performing reads to bypass this technique. But, needless to say, malware authors are leaving no stone unturned to take it to the next level.

Brazilian Banker rootkit

Recently a new rootkit [15] came out that incorporated some very interesting tricks that are worth mentioning.

This rootkit was released for both x64 and x86 platforms. It used developer mode in x64 to install its unsigned drivers to bypass *Windows* driver signing checks. In x64 *Windows* has a special mode that can be turned on for developers for assisting with development. The rootkit used the bcdedit tool to disable integrity checks and enable driver test signing mode in the drivers. The following commands were used:

- cmd /c %tmp%\\bcdedit.exe -set loadoptions DDISABLE_INTEGRITY_CHECKS
- cmd /c %tmp%\\bcdedit.exe -set TESTSIGNING ON

8. Establishing trust

The rootkit installs a boot filter driver with a callback handler using psSetLoadImageNotifyRoutine**.** Whenever a new image is loaded this handler is called. It filters all the other image loads except those of the driver load. Once it detects the driver is about to load it verifies the properties of the driver to decide if it is 'allowed' to load or not. In case the driver is not allowed to load, it overwrites the driver's entry point with a RET instruction. Figure 10 shows the code of the rootkit that writes the error code followed by the RET instruction of a legitimate device driver. Figure 11 shows the code at the EP 0x8f828c05 after it was written.

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

Figure 10: Overwrites bytes at EP.

```
8f828c05 b8010000c0      mov     eax,0C0000001h
8f828c0a c20800          ret     8
8f828c0d dc9185c0b94e    fcom    qword ptr [ecx+4EB9C085h]
```

Figure 11: RET instruction at EP.

The important aspect of this technique is that once the rootkit driver is loaded it ensures that no other security tool can load and disable the threat. Most of the earlier threats actively sought to disable the security products, but this is limited in its approach, since it will only cover known products and versions. Analysis of this rootkit points towards a grim trend where threats establish trust on the essential drivers for the system and everything else could be locked out. Defeating such a trust-based system will be hard since one cannot just rename its files or rehash its binaries to clean out an active infection. Security products will somehow need to break into the trusted circle of the threat.

Popureb

The Popureb rootkit got a lot of media attention after Chun Feng at *Microsoft* blogged about it [16]. This rootkit infects the MBR to gain control of the machine early on in the boot process. It uses the BootRoot technique documented by *eEye* [17] to hook the INT 13 handler and then patch osloader.exe during the boot process to eventually slip its malicious driver into kernel memory instead of beep.sys. Of course, its components like the malicious kernel driver are stored in sectors and are not part of the *Windows* file system. It requires no *Windows* files or registry entries to survive reboot.

This rootkit hooks the DriverStartIO field within the driver object of the lowest driver on the device stack for the hard disk. Using this hook it is able to monitor access to the sectors it wants to protect.

The interesting thing about this rootkit is that, unlike other MBR rootkits, this one does not fake the original MBR image to AV. Instead the malicious MBR is easily readable from user-mode. So once the signature is known, detection of the MBR image is straightforward. But any cross-diff-based tools that generically detect based on discrepancy in the MBR image reads will not raise any alarm.

Even though the Popureb rootkit does not prevent reads, it protects the MBR from writes and makes cleaning challenging. It is not only the prevention of the sector write that makes cleaning challenging. Since Popureb does not fake the clean MBR image, another challenge is to locate the clean MBR to write back. In Popureb's case the clean MBR is ROL encrypted and stored in sectors along with other rootkit components. Its location can be obtained by analysing the malicious MBR disk read code. As noted before, MBR and boot sector infectors will most likely become rampant.

PREDICTING THE NEXT STEALTH SCHEME

In the previous sections we looked at some case studies of various rootkits and the different techniques that are currently being used. Based on the current and previous trend of techniques we can predict the likelihood of their continuation. Precise prediction of which exact kernel object or pointer is going to be the next one is tough. In this section we discuss the overall trends that are likely to continue and need to be tackled from an anti-rootkit perspective. Before we delve into the details of overall trends there is one technique which we thought worth mentioning in this discussion, because of its ease of use.

GDT manipulation to hook IDT on 32-bit OS

During our discussions with security researcher Xeno Kovah at *MITRE Corporation*, he mentioned a technique that he is planning to publish soon and is simple enough for rootkits to adopt. It therefore prompted us to discuss it here (with Xeno's approval) since we think it may be used in the wild in future.

In a yet-to-be-published paper, Xeno explains how the IDT selector value can be manipulated for bypassing various anti-rootkit tools.

This proposed bypass technique is based on the fact that most of the tools only analyse the segment offset field in the IDT table to detect rootkit hooks. The reason such a boundary check usually suffices is because *Windows* uses a flat memory model resulting in only one segment (start: 0x00000000 – end: 0xffffffff). A rootkit would only need to hook this offset field to place its handler in the kernel memory.

Processor 1

	Suspicious	Int	Type	DPL	Present	Segment	Current handler	
1	No	0	IntGate32	0	1	8	82686200	C:\Windows\system32\ntkrnlpa.exe
2	No	1	IntGate32	0	1	8	82686390	C:\Windows\system32\ntkrnlpa.exe
3	No	2	TaskGate	0	1	58	00000000	
4	No	3	IntGate32	3	1	8	82686800	C:\Windows\system32\ntkrnlpa.exe
5	No	4	IntGate32	3	1	8	82686988	C:\Windows\system32\ntkrnlpa.exe
6	No	5	IntGate32	0	1	8	82686ae8	C:\Windows\system32\ntkrnlpa.exe
7	No	6	IntGate32	0	1	8	82686c5c	C:\Windows\system32\ntkrnlpa.exe
8	No	7	IntGate32	0	1	8	82687258	C:\Windows\system32\ntkrnlpa.exe

Figure 12: Tuluka tool showing various fields of IDT.

Processes | Drivers | Devices | SST | GDT | IDT | Sysenter | System threads | Mod

	Suspicious	Selector	Base	Limit	DPL	Type	System	Present	Granul
1	No	0	00000000	00000000	0	Reserved	0	0	0
2	No	8	00000000	FFFFFFFF	0	CodeER_A32	1	1	1
3	No	10	00000000	FFFFFFFF	0	DataRW_A32	1	1	1
4	No	1b	00000000	FFFFFFFF	3	CodeER_A32	1	1	1

Figure 13: Tuluka tool showing various fields in GDT.

However, if a rootkit chooses to manipulate the GDT and modifies an existing index location to be used as a new segment, IDT's segment selector field can be modified while keeping the offset the same.

As an example, Figure 13 shows interrupt gates pointing to segment 8, which translates to index 1 of GDT. For interrupt 5, the calculation at runtime of finding the logical address 0x82686ae8, is derived by adding Base (at GDT index 1) + IDT offset (also called as 'current handler' by Tuluka. So the logical address = 0x0+0x82686ae8.

Now, if the rootkit needs to hook the interrupt 5 without changing the offset it can manipulate the base address at some other GDT index location. So basically two things need to be manipulated:

1. Segment selector in IDT
2. Corresponding index in GDT and its BASE value.

The final pointer would land into the address represented by 'Base' + the 'current offset' in IDT. Thus the hook in the traditional sense won't change the offset value in the IDT, and could bypass some tools.

We feel that although still PoC it would be prudent for anti-rootkit tools to be ready to tackle this technique.

Apart from a one-off specific technique, the following broad areas are likely to be the areas of focus for rootkits and might be used as guidance for product planning:

File forging

The trend of forging file contents has been made mainstream almost single-handedly by the TDSS rootkit as this technique poses significant challenges for traditional rootkit detectors. It is a much more successful option to forge the contents of the file rather than hide them, as hiding files on disk may make them easier targets for rootkit detection tools which have multiple cross diff analysis algorithms to enumerate files on disk and compare them.

Instead of hiding the file, if the contents are forged it makes generic detection a little more challenging, because the anti-rootkit tool will have to match the contents of the file to ensure nothing is changed. Matching the content or calculating even the MD5 of the file is a relatively a more time-consuming process and could slow down scanning speeds. Additionally, with file forging the rootkits usually hide their code in an already existing file, so restoring a file can be challenging or impossible in some cases. We think that this technique will grow more in the coming year.

Memory forging

As noted in the TDSS.E description earlier, we have recently seen the first attempts by malware in the wild to forge kernel memory. Subverting the memory view of a scanner can prove to be very effective for rootkits. Another memory forging technique described via Shadow Walker [14] has been known since 2005.

Due to the complexity of this type of subversion and also due to the fact that solutions exist for known memory forging techniques, their use is likely to be limited. We may see some malware using variations of known techniques by hooking in places not being watched so far. But it is likely to be limited.

Attacks on kernel objects and data variables

Stealth malware are constantly looking to modify places in the kernel that are not being monitored already. The following are some example data structures that are already being attacked and will continue to gain popularity: Device_Object, Driver_Object, KTHREAD, Parseprocedure, KiDebugRoutine and pIofCompleteRequest.

As most major kernel function-pointer tables are now being monitored by anti-rootkit tools the malicious applications are always looking to hook lesser-known kernel variables or hijack dynamic data structures for stealth.

Self-protection and attack

Self-protection is all about the malware watching out for its hooks, detours and infection mechanisms in the system that it is using to gain stealth and stay resident. Any change in its state would immediately be restored by the self-protection code. Rootkits have been using many methods to achieve this, for example TDSS spawns kernel threads to verify that its critical hooks and disk contents are intact. If something is wrong the watcher threads will reinfect the system in an attempt to counter any disinfection by AV. Another variant of TDSS executes a thread which continuously opens exclusive handles on the file it infects in the *Windows* driver directory. So even if its hooks are removed, a typical file open for scanning and repair would not be possible. Privileged access is required in order to close such handles opened by the system process. Since the malware constantly opens these handles, merely closing them is not enough; the thread needs to be disabled as well.

On the notion of 'attack is the best defence', there is also a visible increase in the rootkits that would actively seek to kill AV/anti-rootkit drivers in the kernel. And this trend is likely to continue. Many user-mode attacks are known where malware seeks to kill AV processes. An AV can build pretty solid protection against such user-mode attacks using its kernel drivers in ring-0. But once the malware loads into the kernel, it has an equal opportunity to attack the AV kernel components as well. While protection in ring-0 from ring-0 malware looking to kill AV is not trivial, AV drivers need to continue to raise the bar for malware. The usage of OS registered callbacks is increasing to achieve both self-preservation as well as attacks against AV as seen in both the BlackEnergy and Brazilian Banker rootkits.

Disassociating memory from file on disk

We will continue to see a rise in techniques to disassociate the rootkit file on disk from its module in memory.

Once a rootkit's memory location is identified, for example by following its hooks to the hook-target location, it can give away the rootkit file on disk. By identifying the driver module within which the memory resides the associated file on disk can be revealed. This is a common technique and can be observed in the output of many anti-rootkit tools.

For remediation, the rootkit file on disk needs to be removed so that the rootkit does not become active again on reboot. Obviously this knowledge of associated files is a very useful tool for removing threats.

Memory disassociation can be achieved in many ways; let us look at a few common ones:

- The DKOM-based driver hiding technique as mentioned in [18] can be used to fool tools that just look for drivers using system APIs. Since the driver module will be hidden from the tool, it will not be able to associate the memory address with any module.
- The rootkit can dynamically allocate some memory and copy itself over there, and then install hooks and do all its nefarious work from this newly allocated memory location. Such locations may just appear to be orphans.
- The rootkit can copy itself into some other module's memory. This is similar to the above approach but in this case it would appear that a legitimate module is carrying out the rootkit activity.
- Techniques like those we saw in Koutodoor earlier where a rootkit can delete its file on disk as soon as it loads, and later creates another file and service to load on the next boot – perhaps during a system shutdown. In this case there is no file on disk associated with the active rootkit at any point.

Rootkit authors want to make it harder to remove threats. Disassociating their memory footprint from the file on disk certainly raises the bar. This applies especially to many anti-rootkit tools which rely on their driver to load very early in the boot process for remediation. One advantage of doing so is that the anti-rootkit driver can hopefully load before the rootkit and block other malicious drivers from loading and delete them. For doing this, the anti-rootkit driver needs to know in advance about the malicious drivers to watch out for during boot. One can use known bad hashes but that is not scalable. It is much better to generically detect the rootkit by its activity in memory and save its file location to be used by the anti-rootkit driver during boot. So it is a challenge if the rootkit is using techniques to disassociate (from itself) its files on disk. Rootkits will continue to find new and creative ways to achieve this.

Removing file dependency altogether

Another solution that rootkit authors use to disassociate their file from themselves on disk is not to have a file at all. And while this was limited to exotic threats about a year or two ago, it is becoming more the norm now. We will continue to see a rise in techniques on this front.

Most anti-rootkit tools have capabilities to parse *Windows* file systems and registry hives, and do not rely on *Windows* file-system related code. Therefore they are not defeated by rootkits which work on or above the file-system level because such rootkits look to block file name/location-based access. So no matter which hooks the rootkits use for stealth on or above the file-system level, the anti-rootkit tools would be able to locate hidden objects.

In 2008 we saw the first MBR rootkit which did not require any file to survive on the system. Now there are multiple rootkit and bootkit families such as TDSS.d, Whistler, Fisp and Popureb. These rootkits usually use the MBR infection as a mechanism to load on reboot and do not have their files as part of the *Windows* file system but instead store them on orphan sectors on disk.

Apart from MBR, another technique is to parasitically infect a legitimate driver with the malware's loader. Since the legitimate driver would load when *Windows* boots, the malware's loader would get control. And then it could load its components from disk sectors outside the file system. This technique also has no separate file within the file system for the malware. We may see a decline in rootkits keeping their objects within *Windows* file systems.

Untrusting the trusted

Driver whitelisting and trust-based products are becoming very common nowadays, like *McAfee's Solidcore* [19]; they are even being described as the future of security. More and more applications are getting signed to support the model of trust. Unfortunately, however, it is only a matter of time before rootkits start using the same concept to 'untrust' the security tools, thus locking them out of the system. In the case of the Banker rootkit it is doing virtually the same things that whitelisting-based solutions would do but for its own profit. This trend is likely to grow among kernel malware for locking out their adversaries.

CONCLUSION

This paper has delved into multiple attack scenarios to highlight the current enhancement in the sophistication of stealth attacks. The ease of access to the *Windows* kernel by malicious authors makes the job of writing detection software much tougher. Once the rootkits enter the kernel they seem invincible and they can easily circumvent any and every protection that is in place. Even though specific reactive solutions can be delivered for these rootkits, it seems like a never-ending battle.

The knowledge of the current landscape and the future of stealth could help with finding a solution for both reactive and proactive technology. Most of the current tools and solutions are based on the reactive technologies that find the rootkits after they have installed on the system. Usually they are good enough in rootkit removal once the threat is known; however there can be significant delays in finding out about the rootkit and developing a solution. Developers of anti-rootkit tools can use the prediction section that we have presented previously to assess their technology and reaction time in tackling particular areas.

If proactive detection technologies can be developed that can either catch the rootkit in the act or provide a view of the system that can be trusted, it could be very useful. The challenges discussed in the last section, such as disassociating memory from file on disk or memory forging can be tackled by anti-rootkit tools only when there is a better vantage point from a security perspective. It seems like detection tools may use their ability to intercept rootkit activity during loading using behavioural, trust-based or hypervisor-based systems.

However that might further fuel the arms race much lower than the OS.

REFERENCES

[1] mmpc2. Some Observations on Rootkits. http://blogs.technet.com/b/mmpc/archive/2010/01/07/some-observations-on-rootkits.aspx.

[2] Davis, M.A.; Bodmer, S.; LeMaster, A. Kernel Mode Rootkits. Hacking Exposed Malware & Rootkits: Malware & Rootkits Secrets & Solutions. McGraw-Hill Osborne, 2009.

[3] op., fuzen. Fu Rootkit. http://www.rootkit.com/.

[4] Kapoor, A.; Mathur, R. Challenges in kernel-mode memory scanning. Virus Bulletin International Conference 2009.

[5] Kasslin, K. Kernel Malware: The Attack from within. AVAR 2006.

[6] Kapoor, A.; Mathur, R. McAfee Blogs. May 2007. http://blogs.mcafee.com/mcafee-labs/a-new-rootkid-on-the-block.

[7] mmpc2. October 2008. http://blogs.technet.com/b/mmpc/archive/2008/10/18/uprooting-win32-rustock.aspx.

[8] Prince, B. Russian Banking Trojan BlackEnergy 2 Unmasked at RSA. http://www.eweek.com/c/a/Security/Russian-Banking-Trojan-BlackEnergy-2-Unmasked-at-RSA-883053/.

[9] Ször, P. Personal email communications.

[10] Golovanov, S.; Soumenkov, I. TDL4 – Top Bot. http://www.securelist.com/en/analysis/204792180/TDL4_Top_Bot.

[11] Matrosov, A.; Rodionov, E. The Evolution of TDL: Conquering x64. March 2011. http://www.eset.com/us/resources/white-papers/The_Evolution_of_TDL.pdf.

[12] Mathur, R. Memory Forging Attempt by a Rootkit. McAfee Labs Blog. 21 April 2011. http://blogs.mcafee.com/mcafee-labs/memory-forging-attempt-by-a-rootkit.

[13] Skywing; Skape. KiDebugRoutine. Uninformed. September 2007. http://uninformed.org/index.cgi?v=8&a=2&p=16.

[14] Sparks, S.; Butler, J. "Shadow Walker" Raising the bar for rootkit detection. Black Hat Japan 2005. https://www.blackhat.com/presentations/bh-jp-05/bh-jp-05-sparks-butler.pdf.

[15] Zakorzhevsky, V. An unlikely couple: 64-bit rootkit and rogue AV for MacOS. http://www.securelist.com/en/blog/473/An_unlikely_couple_64_bit_rootkit_and_rogue_AV_for_MacOS.

[16] Feng, C. Don't write it, read it instead! TechNet Blogs. 22 June 2011. http://blogs.technet.com/b/mmpc/archive/2011/06/22/don-t-write-it-read-it-instead.aspx.

[17] Soeder, D.; Permeh, R. BootRoot. 2005. http://www.eeye.com/Resources/Security-Center/Research/Tools/BootRoot.

[18] Hoglund, G.; Butler, J. Rootkits: Subverting The Windows Kernel. Addison-Wesley, 2005.

[19] Solidcore. http://en.wikipedia.org/wiki/Solidcore_Systems.