Dr.WEB®
Anti-virus

Defend what you create

BackDoor.Tdss.565 and its modifications (aka TDL3)

Installation

This piece of malware – a rootkit – presented surprises within minutes after the analysis of its anatomy got underway. For instance, its non-typical method for injection into a system process during installation was something completely unexpected. Though documented, the method has never been implemented in any known virus before and therefore it allows the rootkit to bypass most behaviour blockers, install its driver and yet remain undetected.

Now the installation continues in the kernel mode. The rootkit searches through the stack of devices responsible for interaction with the system disk to determine the driver it is going to infect, its future victim. The choice depends on the hardware configuration. If the system disk uses the IDE interface, it will pick out **atapi.sys**, in other cases it can be **iastor.sys**. There are rootkits that infect file system and network drivers or even the system kernel to ensure their automatic launch (**BackDoor.Bulknet.415**(Virus.Win32.Protector.a/W32/Cutwail.a!rootkit), **Win32.Ntldrbot** (Virus.Win32.Rustock.a/Backdoor:WinNT/Rustock.D), **Trojan.Spambot.2436** (Trojan-Dropper.Win32.Agent.bwg/TR/Drop.Agent.BWG.1) and others) and this instance is not an exception. Note that the file size remains the same as the malicious code is written over a part of the resources section. In fact, the piece of code only occupies 896 bytes (in latter versions it is reduced to 481 byte) and it loads the main body of the rootkit. At the same time it changes the entry point, sets the driver signature link to null and the file's hash sum is recalculated. Addresses of API functions used by the loader for infection are located in its body as RVAs.On one hand it makes the loader much smaller, on the other it complicates analysis of the infected driver in the system that uses a different version of the kernel.

After that the malware assesses available disk space and utilizes its small part (24064 bytes) from the end of the disk for storage of the rootkit's main body or more precisely of that part of the driver which performs the installation saved as binary data instead of an executable image. The block starts with the "**TDL3**" marker followed by 896 bytes of the genuine resources code of the infected driver. It also creates a separate virtual drive where its user mode components and configuration file are located. It looks like this trick was inspired by **BackDoor.Maxplus** (Trojan-Dropper.Win32.Agent.auxo/TROJ_FFSEARCH.A/FFSearcher/Trojan:Win32/Sirefef.A), that also created a virtual disk to deploy its components in the system. Details of the process are described below.

The rootkit's later versions (**BackDoor.Tdss.1030** (Rootkit.Win32.TDSS.y)) store original resources data and their body on the hidden encrypted drive in **rsrc.dat** and **tdl** files respectively, which significantly simplifies its updating.

Upon completion of the installation, the driver returns a **STATUS_SECRET_TOO_LONG (0xC0000154)** error that informs user mode components (http://vms.drweb.com/search/?q=BackDoor.Tdss.565) that installation has completed successfully and makes the system unload the driver that is no longer used by the rootkit.

The loader

The viral loader starts working along with the infected driver. As was already mentioned above, its main task is to load the rootkit's body stored at the «end» of the hard drive. Since the loader starts working when the hard drive port driver is loaded by the kernel, it still can't work with the disk or the file system. That's why it first registers a notification routine for creation of **FS** (FileSystem) control device objects, and only then it loads the rootkit's body. Early versions of the malware used the **IoRegisterFsRegistrationChange** function for this purpose, while the later ones resort to the temporary interception of the victim's **IRP_MJ_DEVICE_CONTROL** in **DRIVER_OBJECT** where the dispatcher waits for a certain request from the file system. Remarkably, in both cases the entry point of the infected driver is used both to start the original DriverEntry as well as for the FS standby (Image 1).

To be consistent, let's assume that **atapi.sys** is the compromised driver.

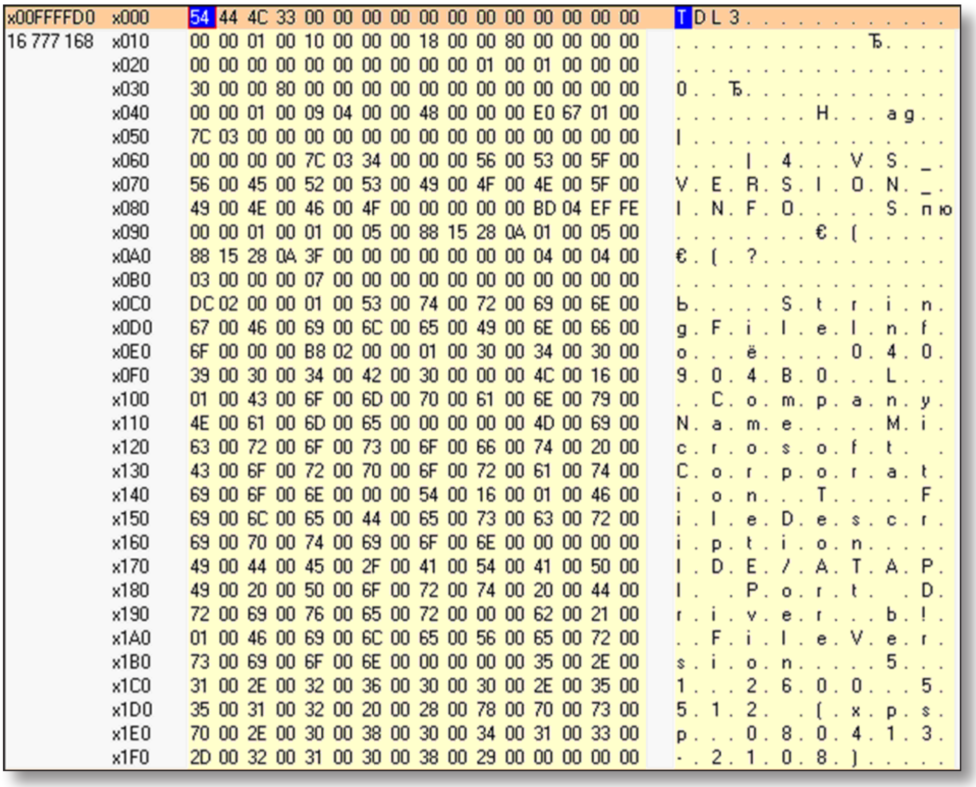
Now let's take a closer look at how the **BackDoor.Tdss.565** (Rootkit.Win32.TDSS.u/Virus:Win32/Alureon.A) loader works. Once it has gained control, it will go over the sections table of its media and modify it to make detection of the initialization section more complicated: nulls the **IMAGE_SCN_MEM_DISCARDABLE** bit of each section, replaces the first byte of a name with zero if it is **INIT**. It also reserves an auxiliary data structure to write the pointer to the **atapi** driver object send to the **DriverEntry** by the kernel. After that it registers using the **CDO** (Control Device Object) **FS** created notification sent to the kernel.

```
.rsrc:00026780                 push    ebp
.rsrc:00026781                 mov     ebp, esp
.rsrc:00026783                 sub     esp, 160h
.rsrc:00026789                 call    $+5
.rsrc:00026789
.rsrc:0002678E                 pop     eax
.rsrc:0002678F                 sub     eax, 0F9A82A0Dh
.rsrc:00026794                 mov     [ebp+var_C], eax
.rsrc:00026797                 mov     eax, [ebp+var_C]
.rsrc:0002679A                 add     eax, 350h
.rsrc:0002679F                 add     eax, 0F9A829FDh
.rsrc:000267A4                 mov     [ebp+pParamBlock], eax
.rsrc:000267A7                 cmp     [ebp+RegistryPath], 1
.rsrc:000267AB                 jbe     jIsFsChangeOccur ; atapi init, not jmp
.rsrc:000267AB
.rsrc:000267B1                 mov     eax, [ebp+DriverObject]
.rsrc:000267B4                 mov     eax, [eax+DRIVER_OBJECT.DriverSection]
.rsrc:000267B7                 mov     eax, [eax]
.rsrc:000267B9                 mov     [ebp+PsLoadedModuleList], eax
.rsrc:000267B9
.rsrc:000267BC
.rsrc:000267BC jNextDriverLdrEntry:                    ; CODE XREF: DriverEntry+4F↓j
.rsrc:000267BC                 mov     eax, [ebp+PsLoadedModuleList]
.rsrc:000267BF                 movzx   eax, [eax+KLDR_DATA_TABLE_ENTRY.LoadCount]
.rsrc:000267C3                 test    eax, eax
.rsrc:000267C5                 jz      short jPsLoadedModuleListWasFound
```

Image 1.

The entry point of atapi.sys compromised by **BackDoor.Tdss.565**

As the file system request is received, the second part of the loader is started. It checks all object-devices of the port driver (e.g., **"\Device\IdePort0"**, **"\Device\IdeDeviceP0T0L0-3"**) and uses the disk offset placed in its body during the installation to read the rootkit's body. Though the method where the ordinary **ZwOpenFile**, **ZwReadFile** functions are used for this purpose seems not quite sophisticated since the malware has to check devices one by one, it allows the loader to remain compact and serves its purpose quite well. The TDL3 signature placed at the beginning of the data segment is used to verify if the reading has been successful (Image 2). After that the notification is deleted (**IoUnregisterFsRegistrationChange**) and control is transferred to the body of the rootkit.



```
x00FFFFD0  x000  54 44 4C 33 00 00 00 00 00 00 00 00 00 00 00 00  TDL3............
16 777 168 x010  00 00 01 00 10 00 00 00 18 00 00 80 00 00 00 00  ............Ђ....
           x020  00 00 00 00 00 00 00 00 00 00 01 00 01 00 00 00  ................
           x030  30 00 00 80 00 00 00 00 00 00 00 00 00 00 00 00  0..Ђ............
           x040  00 00 01 00 09 04 00 00 48 00 00 00 E0 67 01 00  ........H...ag..
           x050  7C 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |...............
           x060  00 00 00 00 7C 03 34 00 00 00 56 00 53 00 5F 00  ....|.4...V.S._.
           x070  56 00 45 00 52 00 53 00 49 00 4F 00 4E 00 5F 00  V.E.R.S.I.O.N._.
           x080  49 00 4E 00 46 00 4F 00 00 00 00 00 BD 04 EF FE  I.N.F.O.....S.пю
           x090  00 00 01 00 01 00 05 00 88 15 28 0A 01 00 05 00  ........€.(.....
           x0A0  88 15 28 0A 3F 00 00 00 00 00 00 00 04 00 04 00  €.(.?...........
           x0B0  03 00 00 00 07 00 00 00 00 00 00 00 00 00 00 00  ................
           x0C0  DC 02 00 00 01 00 53 00 74 00 72 00 69 00 6E 00  Ь.....S.t.r.i.n.
           x0D0  67 00 46 00 69 00 6C 00 65 00 49 00 6E 00 66 00  g.F.i.l.e.I.n.f.
           x0E0  6F 00 00 00 B8 02 00 00 01 00 30 00 34 00 30 00  o...ё.....0.4.0.
           x0F0  39 00 30 00 34 00 42 00 30 00 00 00 4C 00 16 00  9.0.4.B.0...L...
           x100  01 00 43 00 6F 00 6D 00 70 00 61 00 6E 00 79 00  ..C.o.m.p.a.n.y.
           x110  4E 00 61 00 6D 00 65 00 00 00 00 00 4D 00 69 00  N.a.m.e.....M.i.
           x120  63 00 72 00 6F 00 73 00 6F 00 66 00 74 00 20 00  c.r.o.s.o.f.t. .
           x130  43 00 6F 00 72 00 70 00 6F 00 72 00 61 00 74 00  C.o.r.p.o.r.a.t.
           x140  69 00 6F 00 6E 00 00 00 54 00 16 00 01 00 46 00  i.o.n...T.....F.
           x150  69 00 6C 00 65 00 44 00 65 00 73 00 63 00 72 00  i.l.e.D.e.s.c.r.
           x160  69 00 70 00 74 00 69 00 6F 00 6E 00 00 00 00 00  i.p.t.i.o.n.....
           x170  49 00 44 00 45 00 2F 00 41 00 54 00 41 00 50 00  I.D.E./.A.T.A.P.
           x180  49 00 20 00 50 00 6F 00 72 00 74 00 20 00 44 00  I. .P.o.r.t. .D.
           x190  72 00 69 00 76 00 65 00 72 00 00 00 62 00 21 00  r.i.v.e.r...b.!.
           x1A0  01 00 46 00 69 00 6C 00 65 00 56 00 65 00 72 00  ..F.i.l.e.V.e.r.
           x1B0  73 00 69 00 6F 00 6E 00 00 00 00 00 35 00 2E 00  s.i.o.n.....5...
           x1C0  31 00 2E 00 32 00 36 00 30 00 30 00 2E 00 35 00  1...2.6.0.0...5.
           x1D0  35 00 31 00 32 00 20 00 28 00 78 00 70 00 73 00  5.1.2. .(.x.p.s.
           x1E0  70 00 2E 00 30 00 38 00 30 00 34 00 31 00 33 00  p...0.8.0.4.1.3.
           x1F0  2D 00 32 00 31 00 30 00 38 00 29 00 00 00 00 00  -.2.1.0.8.).....
```

Image 2.

The first sector of the rookit's body located in end sectors of the hard drive

The rootkit

Surely, an encrypted drive with its own file system is among most notable technical features of **TDL3**. The mechanism used to hide the entire file or a part of an arbitrary disk sector on the port driver level is remarkable too. No other known rootkit has the concepts implemented in full.

It is well known that the main feature of the NT virtual file system is availability of all input-output devices on the descriptor layer where the key element is the file object created by the kernel and objects that represent the device. An application opens the descriptor for one channel, hard drive, volume or file and different layers of input-output devices stack participate in the interaction. The kernel only needs information about a request and starts a corresponding dispatcher function.

Authors of the rootkit used a similar approach and implemented their file system working on the level of device object's port driver so that the virus mounts its **FS** to the device object.

The **atapi** driver creates several types of device objects (Image 3). The upper two are devices representing hard and CD drives while the other two are controllers interacting with the mini-port driver implemented in Windows XP as a hybrid mix of a port and a mini-port. To mount its hidden drive, the rootkit chooses a device object with the **FILE_DEVICE_CONTROLLER** type.

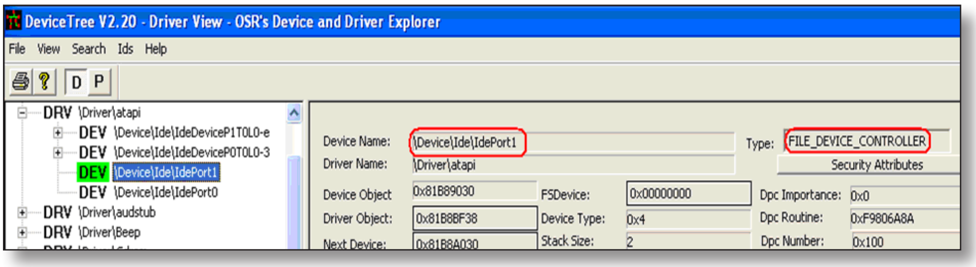


Image 2.
Devices created by **atapi.sys**

An ordinary ("healthy") **atapi** uses only one IRP dispatch function to serve read/write requests – **IRP_MJ_SCSI (IRP_MJ_INTERNAL_DEVICE_CONTROL)**. The client uses **Srb** and sends it to the disk device object. **SUCCESS** is always returned for **Create/Close atapi** requests since the **atapi** doesn't use them. However, the Create operation is very important for **FSD** (File System Driver) since it initializes **FILE_OBJECT** used for file operations.

The path to rootkit files located in the protected (hidden) area looks as follows:

\Device\Ide\IdePort1\mjqxtpex, where **mjqxtpex** is a 8-byte signature generated randomly at system startup. The hidden drive is used by user mode components of the rootkit to store their files received from the Internet or to read their configuration.

Full path example:

```
\\?\globalroot\Device\Ide\IdePort1\mjqxtpex\tdlcmd.dll
\\?\globalroot\Device\Ide\IdePort1\mjqxtpex\tdlwsp.dll
\\?\globalroot\Device\Ide\IdePort1\mjqxtpex\config.ini
```

In order to understand how the rootkit works with its file system, let's take a look at the flow-chart that shows how a create request is normally processed (**ntfs** or **fast-fat**) and see how **\Device\HarddiskVolume1\directory\config.ini** is opened on an ordinary drive and how – **\Device\Ide\IdePort1\mjqxtpex\config.ini** is accessed on the hidden drive (Image 4).

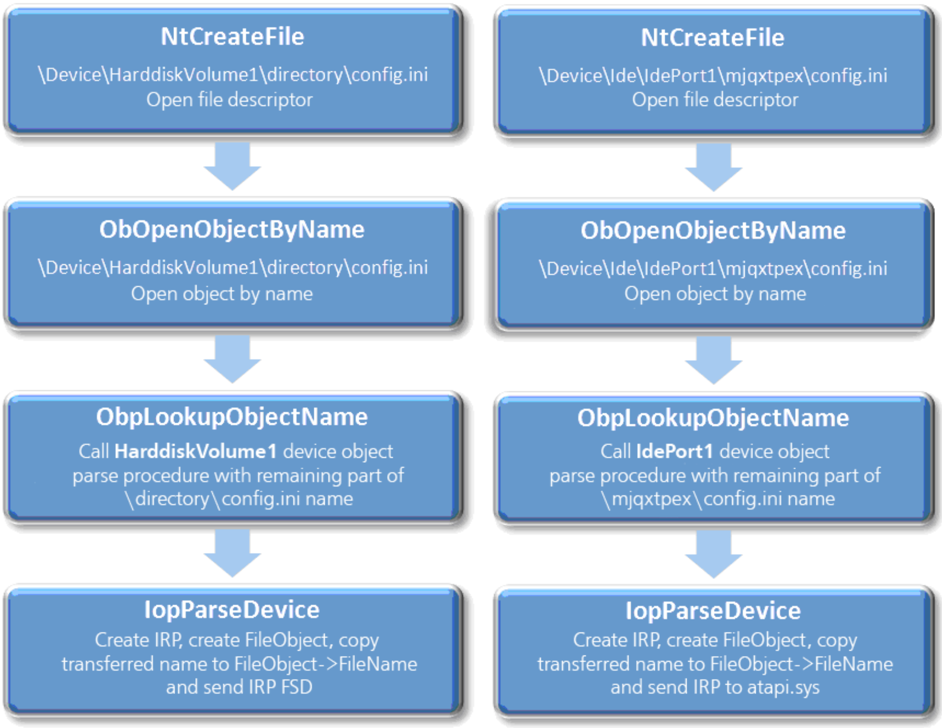


Image 4.

Opening a file on an ordinary disk drive and opening a file on the hidden drive

The rootkit has one shared dispatch function for all requests from **atapi**, clients and user mode components. Therefore it performs two important tasks:

- Hides data located in the protected area from **atapi** clients and provides clients with an original file as they try to read data from the disk.
- As with **FSD** it handles **create/close/query information** request for files from the protected area sent by roootkit's **dll** used by processes as well as from the rootkit itself that may request to read a section of **config.ini**.

The rootkit replaces parameters in the dispatch functions pointer table as follows: it finds the end of the first section of **atapi.sys** file in the memory and writes the following template into the **cave** (the remaining free space in the section):

```
mov eax, ds:0FFDF0308h
jmp dword ptr [eax+0FCh]
```

In some cases the instructions can overwrite data in the adjacent section since there is no any verification procedure. Therefore interceptions are still directed to **atapi.sys** (Image 5). It deceives many anti-rootkits so the malware remains undetected.

```
Dispatch routines:
[00] IRP_MJ_CREATE                      f9756b3a      atapi!PortPassThroughZeroUnusedBuffers+0x34
[01] IRP_MJ_CREATE_NAMED_PIPE           f9756b3a      atapi!PortPassThroughZeroUnusedBuffers+0x34
[02] IRP_MJ_CLOSE                       f9756b3a      atapi!PortPassThroughZeroUnusedBuffers+0x34
[03] IRP_MJ_READ                        f9756b3a      atapi!PortPassThroughZeroUnusedBuffers+0x34
[04] IRP_MJ_WRITE                       f9756b3a      atapi!PortPassThroughZeroUnusedBuffers+0x34
[05] IRP_MJ_QUERY_INFORMATION           f9756b3a      atapi!PortPassThroughZeroUnusedBuffers+0x34
[06] IRP_MJ_SET_INFORMATION             f9756b3a      atapi!PortPassThroughZeroUnusedBuffers+0x34
[07] IRP_MJ_QUERY_EA                    f9756b3a      atapi!PortPassThroughZeroUnusedBuffers+0x34
[08] IRP_MJ_SET_EA                      f9756b3a      atapi!PortPassThroughZeroUnusedBuffers+0x34
[09] IRP_MJ_FLUSH_BUFFERS               f9756b3a      atapi!PortPassThroughZeroUnusedBuffers+0x34
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION    f9756b3a      atapi!PortPassThroughZeroUnusedBuffers+0x34
[0b] IRP_MJ_SET_VOLUME_INFORMATION      f9756b3a      atapi!PortPassThroughZeroUnusedBuffers+0x34
[0c] IRP_MJ_DIRECTORY_CONTROL           f9756b3a      atapi!PortPassThroughZeroUnusedBuffers+0x34
```

Image 5.
Windows XP SP3 atapi.sys interceptions

The rootkit utilizes a large structure for storage of all configuration information that may be required to perform its routines. The structure pointer is placed at **0xFFDF0308**, i.e. a part of **KUSER_SHARED_DATA** is used. The request dispatcher is found at the +**00FCh** offset (invoked in the example above – `jmp dword ptr [eax+0FCh]`). Structures describing which sectors must be hidden and what should replace them are also stored there.

If an **atapi** client requests data from the protected drive, it will simply zero-fill it or replace it with original data. Let's take a look at the pseudo code showing how it works:

```
if( DeviceObject == ROOTKIT _ PARAM _ BLOCK. AtapiBoot-
RootkitDevObj &&
    IoStack->MajorFunction == IRP _ MJ _ SCSI &&
    IoStack->Parameters.Scsi.Srb->Function == SRB _ FUNC-
TION _ EXECUTE _ SCSI
)
{
if( RequestedStartSector + cSectors > ROOTKIT _ PARAM _
BLOCK.HideAreaStartSector)
{
       if( IsRead )
       {
              Replace the completion function of the
current stack location with its own function
       }
       else if( IsWrite )
       {
              End operation and return an error
}
else if( a request to the atapi or oep resource section,
chksum,
security data dir entry)
{
Replace the completion function of the current stack
element with its own
function
}
}
```

So it is the completion function where the data is replaced.

Once the first versions of TDL3 were found in the wild, some developers of anti-rootkit software made corresponding changes in to their products so that they would at least detect the rootkit. Virus makers were quick to reply and created new versions of the malware featuring new interception techniques which are harder to detect.

Now the dispatch table of the compromised driver remains clean. Authors of the rootkit used a non-standard approach. They simply "stole" from the **atapi** the device object working with the system drive they are going to use (Image 6).

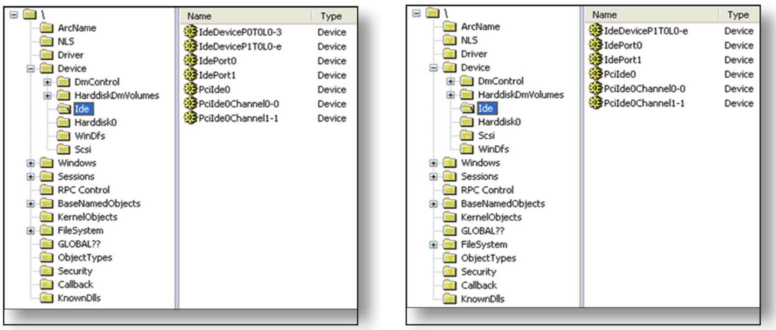
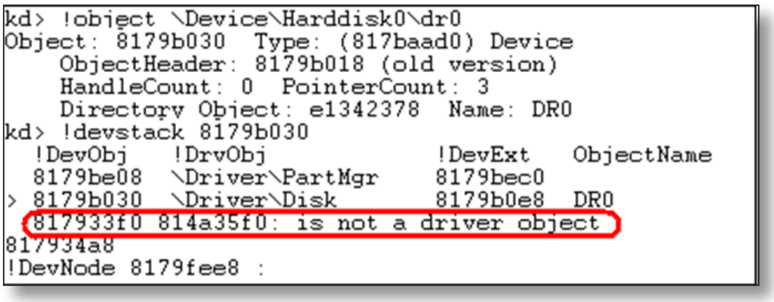


Image 6.

Clean system (on the left) and infected system (on the right) with the device "missing"

The abnormality can only be detected with a debugger (Image 7) – an unknown device using an unknown driver. Moreover, the **DRIVER_OBJECT** header of the "unknown driver" is corrupt while the driver is removed from the system drivers list (and the "stolen device" too). The driver object is created by the rootkit to hide sectors of the hard drive and provide access to the hidden sectors for the malware. It has already become visible but you still need to find or guess a device with a name comprised of 8 random characters.



```
kd> !object \Device\Harddisk0\dr0
Object: 8179b030  Type: (817baad0) Device
    ObjectHeader: 8179b018 (old version)
    HandleCount: 0  PointerCount: 3
    Directory Object: e1342378  Name: DR0
kd> !devstack 8179b030
  !DevObj   !DrvObj            !DevExt   ObjectName
  8179be08  \Driver\PartMgr    8179bec0
> 8179b030  \Driver\Disk       8179b0e8  DR0
  817933f0 814a35f0: is not a driver object
817934a8
!DevNode 8179fee8 :
```

Image 7.

Detecting the abnormality with WinDbg

Now developers of anti-rootkits will have to devise a new way of how to use a specified device object to find a real driver used by the device. The debug output of the rootkit upon its launch is also quite unusual. It reveals passion of the virus makers for cartoons. For instance, it can display one of the following lines:

- Spider-Pig, Spider-Pig, does whatever a Spider-Pig does. Can he swing, from a web? No he can't, he's a pig. Look out! He is a Spider-Pig!
- This is your life, and it's ending one minute at a time
- The things you own end up owning you
- You are not your fucking khakis

And in the later versions:

- Alright Brain, you don't like me, and I don't like you. But let's just do this, and I can get back to killing you with beer
- I'm normally not a praying man, but if you're up there, please save me Superman.
- Dude, meet me in Montana XX00, Jesus (H. Christ)
- Jebus where are you? Homer calls Jebus!
- TDL3 is not a new TDSS!

The rootkit file system

At the end of the hard drive the rootkit occupies a certain area utilized to store its body and the virtual drive. The structure of a physical drive in a compromised system looks as follows:

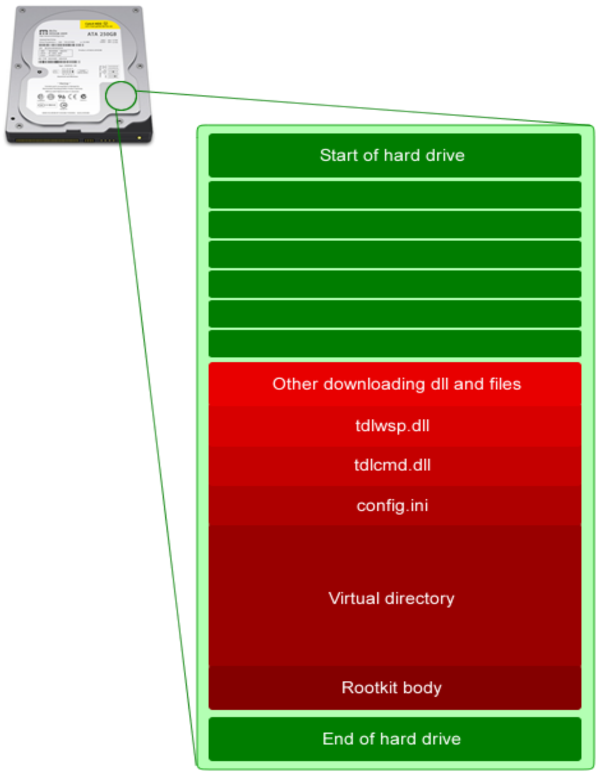


Image 8.
Rootkit file system

Sector numbers of the virtual drive increase from the upper sectors to lower ones and the rootkit uses the negative offset starting from the sector utilized as a descriptor of the virtual directory (Image 9). So expanding backwards it can overwrite data in other sectors of the physical drive.

File metadata and other information are placed in one file in the hidden disk drive. The metadata size is 12 bytes and it has the following format:

```
+00 Signature [TDLD – a directory, TDLF – a file, TDLN – a file
from the Internet]
+04 an ordinal number of a sector with valid data
+08 data size, if the sector provides sufficient space for
storage or if zero is not set for the preceding field, the
offset from  file data to the next sector where the file code
is stored (i.e. +0xC for metadata, so the field usually con-
tains 0x3F4, 0x3F4 + 0xC = 0x400)
```

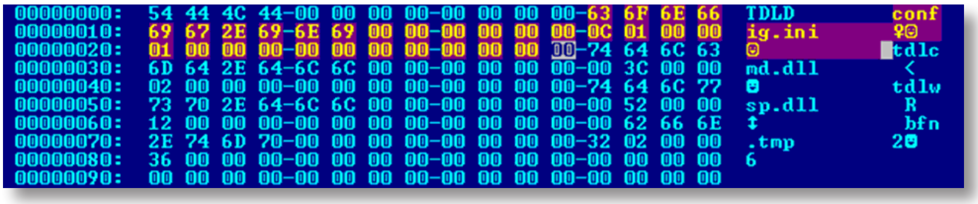


Image 9.

BackDoor.Tdss.565 virtual directory descriptor

On the Image 9, you can see three files written onto the disk during the rootkit installation (**config.ini**, **tdlcmd.dll** и **tdlwsp.dll**) and the **bfn.tmp** temporary file downloaded from the Internet. All sectors locating the drive are encrypted using **RC4**. The same encryption algorithm is used by other components that are not involved in operation of the file system. The file described above is encrypted using the bot ID stored in **config.ini**. After decryption it appears as a set of commands for the rootkit (Image 10).

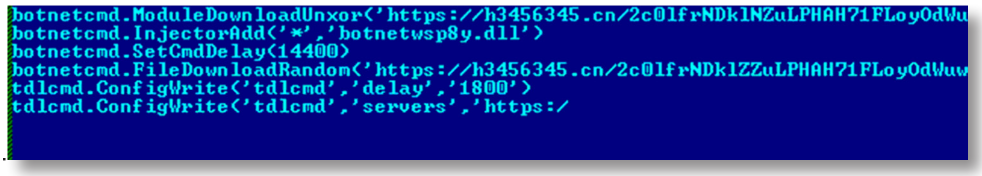


Image 10.

Contents of bfn.tmp

On the Image 11, you can see a descriptor for the **BackDoor.Tdss.1030** directory. Here you can find new file metadata fields and data data for separate files of the rootkit body (**tdl**) and original resources of the infected file (**rsrc.dat**):

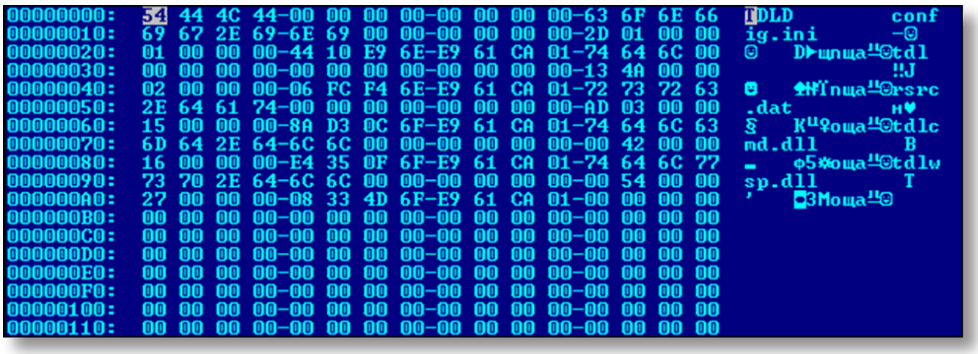


Image 11.

BackDoor.Tdss.1030 virtual directory descriptor

The directory incorporates a metadata structure and subsequent file entries. The size of each entry is 32 bytes (an entry on the Image 9 is highlighted).

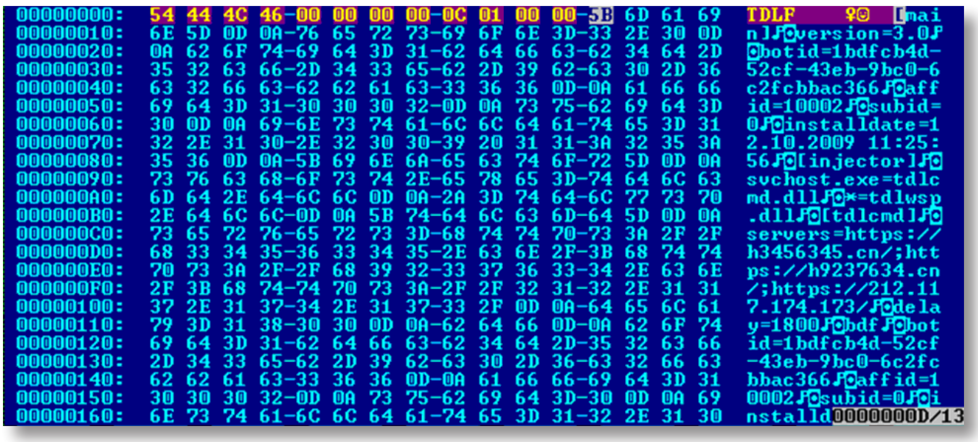


Image 12.

File descriptor

First 12 bytes of the file descriptor contain metadata with the **TDLF** or **TDLN** signature, the number of the next sector and size placed at the beginning. For instance, on the Image 13 you can see that the specified file size is 0x10C bytes.

In the rootkit's file system, a sector containing data is followed by a "trash" sector since the rootkit works with 0x400 bytes units (Image 13) instead of 0x200 (for standard systems).

```
                                    ; int __stdcall fnReadDataFromProtectedAreaAndDecryptIt(int Buffer,
                                    fnReadDataFromProtectedAreaAndDecryptIt proc near
                                                                                   ; CODE XREF: sub_8198916C+2
                                                                                   ; fnVFSReadFsRecord_Central

                                    Buffer              = dword ptr  8
                                    DataTransferLength= dword ptr  0Ch
                                    pNegativeOffsFromTail= dword ptr  10h

000 55                                                push    ebp
004 8B EC                                             mov     ebp, esp
004 A1 08 03 DF FF                                    mov     eax, ds:0FFDF0308h
004 8B 4D 10                                          mov     ecx, [ebp+pNegativeOffsFromTail]
004 53                                                push    ebx
008 8B 98 08 01 00 00                                 mov     ebx, [eax+108h] ; sector size
008 56                                                push    esi
00C 8B 70 10                                          mov     esi, [eax+10h]
00C 2B 31                                             sub     esi, [ecx+ULARGE_INTEGER.u.LowPart]
00C 57                                                push    edi
010 8B 78 14                                          mov     edi, [eax+14h]
010 1B 79 04                                          sbb     edi, [ecx+ULARGE_INTEGER.u.HighPart]
010 81 EE 00 04 00 00                                 sub     esi, 400h
010 68 6F 18 7C E4                                    push    _alldiv         ; FunctionChksum
014 83 DF 00                                          sbb     edi, 0
014 E8 76 E4 FF FF                                    call    fnFindNtoskrnlStart

014 50                                                push    eax             ; pImage
018 E8 2F E5 FF FF                                    call    fnGetSystemRoutineAddress

010 33 C9                                             xor     ecx, ecx
010 51                                                push    ecx
014 53                                                push    ebx
018 57                                                push    edi
01C 56                                                push    esi
020 FF D0                                             call    eax             ; alldiv
```

Image 13.

Reading sectors
of the virtual drive

Conclusion

All in all, new **BackDoor.Tdss** rootkits are sophisticated pieces of malware. Their detection and neutralization pose a serious challenge to anti-virus vendors. And as it has already happened **BackDoor.MaosBoot** (Mebroot), **Win32.Ntldrbot** (Rustock.C) and other rootkits not all vendors rise to it.

3d street Yamskogo polya 2-12A
Moscow, Russia 125124

Phone: +7 (495) 789-45-87

Fax: +7 (495) 789-45-97

www.drweb.com | www.freedrweb.com | www.av-desk.com