

# TDL4 Analysis Paper: a brief introduction and How to Debug It

Hi all!

I would like to present the result of my work on this threat. After I have developed an anti TDL 3 System I analyzed this relative new threat written by the same authors of TDL3 for X64 Systems. Because there aren't much analysis paper on internet about this rootkit (or exactly nothing of them that explains how to debug it), I decided to wrote my own paper. I'm an Italian security researcher, English is not my native language, indeed if the reader found some language error, please send me an email ([aall86@altermista.org](mailto:aall86@altermista.org)).

## Introduction

First of all I need to warn the reader that this guide is about TDL4 new features, it doesn't cover any of old TDL3 peculiarity, like port driver infection and many others... The reader can find all the information using these others analysis papers:

- <http://www.eset.com/resources/white-papers/TDL3-Analysis.pdf> (very useful one)
- <http://www.securelist.com/en/analysis/204792131/TDSS>
- <http://support.cmclab.net/vn/index.php/topic,6934.0.html>

TDL4, unlike its predecessor, uses a completely different installation method. It infect system Mbr and is fully compatible with x64 systems. Infection is fully encrypted, begin in Mbr (with a simple 20 bytes encryption method), and in its file system (that uses a RC4 encryption method with physical sector numbers used as encryption key).

## FORGED MBR

The rootkit mbr is quite simple, its only job is to read Rootkit "ldr16" file from its encrypted file system and transfer control to it. First it read last sector of the current bootable hard disk, with the aid of interrupt 0x13 (function code 0x48 to get total hard disk sector and function code 0x42 to perform the actual reading). Last sector contains rootkit file table. Mbr program decrypt file table with its own code, then from file table it gets "ldr16" sectors offset and it reads and decrypt them, remapping entire file in a specific memory address... It finally transfer control to loader file. Actual int 13 hook is implemented in "ldr16" file.

```

mov     ds:7B2h, dl      ; 0x1b2 = buffer che memorizza il drive Index
sub     word ptr ds:413h, 10h
mov     ax, ds:413h
shl     ax, 6           ; AX = Segmento dove viene copiato l'intero ldr16
mov     ds:674h, ax
mov     ah, 48h ; 'H'   ; Read Drive Parameters
mov     si, 8C5h       ; 0000:08c5 -> Result buffer
mov     word ptr ds:8C5h, 1Eh ; Set buffer size
int     13h           ; DISK -
mov     di, 758h       ; di = "ldr16" string (relative pos: 0x15B)

```

Figure 1. The ldr16 disk geometry read approach

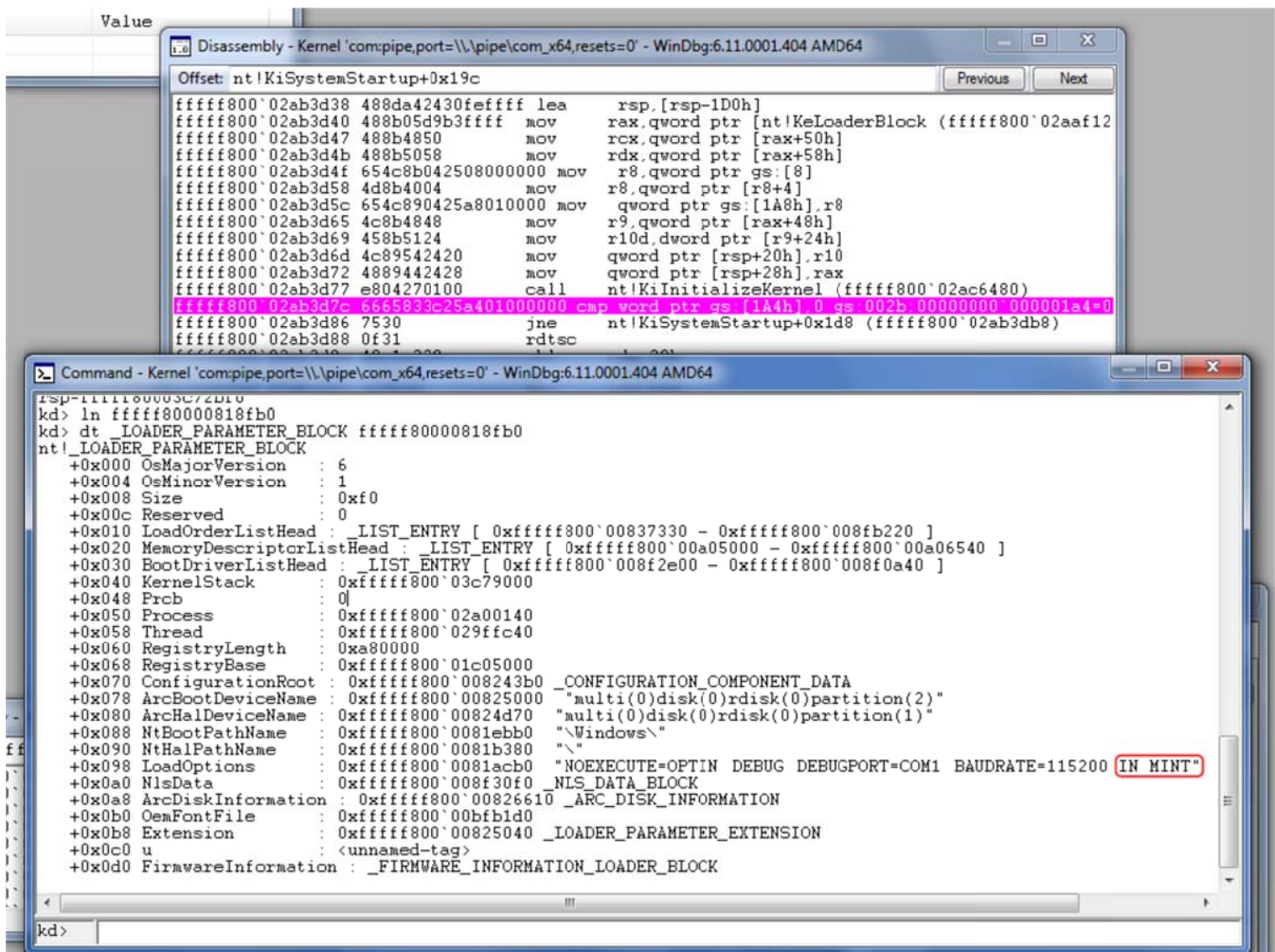
## 16 BIT LOADER

"ldr16" file is the rootkit 16 bit loader. It's a binary file that contains int13 hook, anti driver-signing enforcement strategy and "kdcom.dll" replacement code. "ldr16" begins its execution from mbr. It first read and decrypt original mbr (and map it in canonical memory address 0000:7c00) from encrypted rootkit file system, in a simil-manner as rootkit mbr. It then install its int13 hook routine and finally releases control to the original system mbr. Windows mbr and boot code run as nothing was modified, not become aware of alteration.

I have released a big guide on this infection approach for x64 systems available here:

[http://www.aall86.altervista.org/guide/X64\\_MBR\\_Rootkits.pdf](http://www.aall86.altervista.org/guide/X64_MBR_Rootkits.pdf) (unfortunately is in italian language, if the reader is interested on topic, he can use Google to translate it).

In this way the rootkit loader monitor all disk read, and, if it recognize "BcdLibraryBoolean\_EmsEnabled" startup boot configuration data flag, it switches to "BcdOSLoaderBoolean\_WinPEMode" one, with the objective to disable Loader Integrity Check Enforcement (that doesn't load if the system is in WinPe mode, the reader would like to check "OslInitializeCodeIntegrity" winload function). To restore the original normal environment, after the fake rootkit "kdcom.dll" was loaded (we will see afterward how the replacement is done), and before Winload releases control to Nt Kernel, the rootkit switches back the "/MININT" loader block flag with a senseless "IN/MINT" flag. As result Windows boot process proceed regular in normal mode...



```
Disassembly - Kernel 'com:pipe,port=\\.\pipe\com_x64, resets=0' - WinDbg:6.11.0001.404 AMD64
Offset: nt!KiSystemStartup+0x19c
fffff800`02ab3d38 488da42430feffff lea   rsp,[rsp-1D0h]
fffff800`02ab3d40 488b05d9b3ffff mov   rax,qword ptr [nt!KeLoaderBlock (fffff800`02aaf12
fffff800`02ab3d47 488b4850      mov   rcx,qword ptr [rax+50h]
fffff800`02ab3d4b 488b5058      mov   rdx,qword ptr [rax+58h]
fffff800`02ab3d4f 654c8b042508000000 mov  r8,qword ptr gs:[8]
fffff800`02ab3d58 4d8b4004      mov   r8,qword ptr [r8+4]
fffff800`02ab3d5c 654c890425a8010000 mov  qword ptr gs:[1A8h],r8
fffff800`02ab3d65 4c8b4848      mov   r9,qword ptr [rax+48h]
fffff800`02ab3d69 458b5124      mov   r10d,dword ptr [r9+24h]
fffff800`02ab3d6d 4c89542420    mov  qword ptr [rsp+20h],r10
fffff800`02ab3d72 4889442428    mov  qword ptr [rsp+28h],rax
fffff800`02ab3d77 e804270100    call nt!KiInitializeKernel (fffff800`02ac6480)
fffff800`02ab3d7c 6665833c25a401000000 cmp  word ptr gs:[1A4h],0 gs_002b_00000000_000001a4=0
fffff800`02ab3d86 7530          jne   nt!KiSystemStartup+0x1d8 (fffff800`02ab3db8)
fffff800`02ab3d88 0f31         rdtsc

Command - Kernel 'com:pipe,port=\\.\pipe\com_x64, resets=0' - WinDbg:6.11.0001.404 AMD64
!sp-111100003c72d10
kd> ln fffff80000818fb0
kd> dt _LOADER_PARAMETER_BLOCK fffff80000818fb0
nt!_LOADER_PARAMETER_BLOCK
+0x000 OsMajorVersion : 6
+0x004 OsMinorVersion : 1
+0x008 Size : 0xf0
+0x00c Reserved : 0
+0x010 LoadOrderListHead : _LIST_ENTRY [ 0xfffff800`00837330 - 0xfffff800`008fb220 ]
+0x020 MemoryDescriptorListHead : _LIST_ENTRY [ 0xfffff800`00a05000 - 0xfffff800`00a06540 ]
+0x030 BootDriverListHead : _LIST_ENTRY [ 0xfffff800`008f2e00 - 0xfffff800`008f0a40 ]
+0x040 KernelStack : 0xfffff800`03c79000
+0x048 Prcb : 0
+0x050 Process : 0xfffff800`02a00140
+0x058 Thread : 0xfffff800`029ffc40
+0x060 RegistryLength : 0xa80000
+0x068 RegistryBase : 0xfffff800`01c05000
+0x070 ConfigurationRoot : 0xfffff800`008243b0 _CONFIGURATION_COMPONENT_DATA
+0x078 ArcBootDeviceName : 0xfffff800`00825000 "multi(0)disk(0)rdisk(0)partition(2)"
+0x080 ArcHalDeviceName : 0xfffff800`00824d70 "multi(0)disk(0)rdisk(0)partition(1)"
+0x088 NtBootPathName : 0xfffff800`0081ebb0 "\\Windows\\"
+0x090 NtHalPathName : 0xfffff800`0081b380 "\\\"
+0x098 LoadOptions : 0xfffff800`0081acb0 "NOEXECUTE=OPTIN DEBUG DEBUGPORT=COM1 BAUDRATE=115200 IN MINT"
+0x0a0 NlsData : 0xfffff800`008f30f0 _NLS_DATA_BLOCK
+0x0a8 ArcDiskInformation : 0xfffff800`00826610 _ARC_DISK_INFORMATION
+0x0b0 OemFontFile : 0xfffff800`00bfb1d0
+0x0b8 Extension : 0xfffff800`00825040 _LOADER_PARAMETER_EXTENSION
+0x0c0 u : <unnamed-tag>
+0x0d0 FirmwareInformation : _FIRMWARE_INFORMATION_LOADER_BLOCK
```

Figure 2. Infected machine's Nt Loader block. It's noteworthy "LoadOptions" field that highlights "IN MINT" rootkit options

As we expected, If the filtered disk read procedure found "kdcom.dll" library reading, it replace real library with the rootkit one, which is called "ldr32/64" (depending on architecture of the operation system). But how rootkit recognize "kdcom.dll" (the kernel debugger main library)?

The answer is simple, ldr16 recognize kdcom.dll analyzing its PE Header. It first looks at Dos Header, searching for signature 0x5A4D ('MZ' signature), then it looks at PE header signature 0x4550 ('PE'). At this time it determines if PE is in 32 bit or 64 bit format, and then **it checks Export Directory Size**.

```

seg000:0109 SearchFile:                                ; CODE XREF: seg000:00FD↑j
seg000:0109                                ; seg000:0105↑j
seg000:0109      cmp     word ptr es:[bx], 5A4Dh
seg000:010E      jnz     loc_20E
seg000:0112      mov     di, es:[bx+3Ch] ; DI = dosHdr->e_lfaNew
seg000:0116      cmp     word ptr es:[bx+di], 4550h ; PE
seg000:011B      jnz     loc_20E
seg000:011F      cmp     word ptr es:[bx+di+18h], 100h ; Is this PE a 32 bit version? (Flag 0x10B)
seg000:0125      jnz     short ReplaceKdCom64
seg000:0127      cmp     dword ptr es:[bx+di+7Ch], 0FAh ; ' ' ; if (ntHdr.OptHdr.DataDirectory.
seg000:0130      jnz     loc_20E                                ExportDirSize != 0xFA)
seg000:0134      mov     si, 413h ; SI = "ldr32"
seg000:0137      mov     cx, 6
seg000:013A      jmp     short ReadFile
seg000:013C ;
seg000:013C
seg000:013C ReplaceKdCom64:
seg000:013C      cmp     dword ptr es:[bx+di+8Ch], 0FAh ;
seg000:0146      jnz     loc_20E
seg000:014A      mov     si, 419h ; SI = "ldr64"
seg000:014D      mov     cx, 6

```

Figure 3. "Kdcom.dll" ldr16 identification Code

If it discovered that Export Directory size is equal to kdcom's one, it has found kdcom.dll and begins to replace its content causing Windbg to fail connect to the infected host.

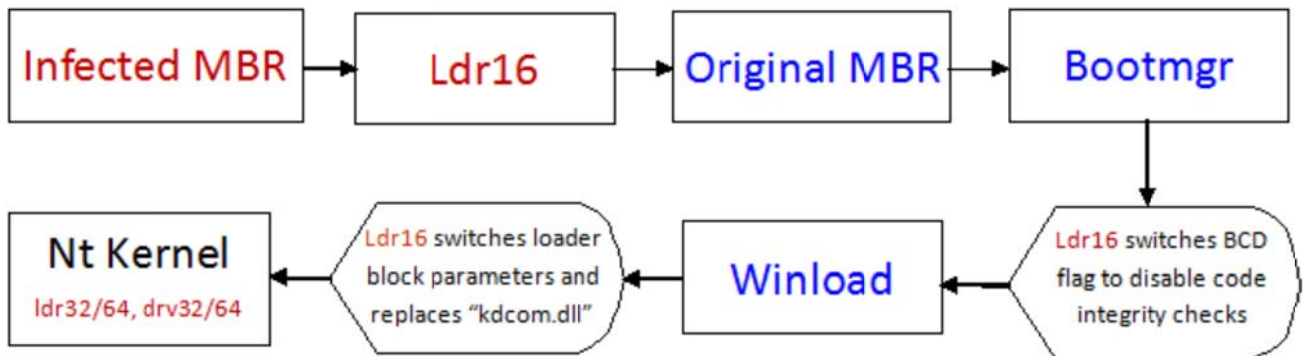


Figure 4. TDL forged Master boot record and Ldr16 execution flow  
Red color is for Rootkit component, blue is for original OS components

Now it's time to take a glance at real infected driver loader, the fake rootkit "kdcom.dll"...

## 32/64 BIT LOADER

In my own opinion, the most complex and clever part of the infection is the "ldr32/64" rootkit loader. From now we'll refer this part of rootkit as "driver loader", because it's target is to load main infected driver. This file is actually a dll that export all of the real original "kdcom.dll" functions. All exported functions do nothing (in this way the debugging is impossible), except one, KdDebuggerInitialize1. This special procedure loads and execute rootkit main kernel driver (its file name is "drv32/64" depends on system architecture like "ldr32/64").

```

public KdDebuggerInitialize1
KdDebuggerInitialize1 proc near          ; DATA XREF: .text:off_180001088f0
    lea    rcx, TDLLoadNotifyRoutine
    jmp    cs:PsSetCreateThreadNotifyRoutine
KdDebuggerInitialize1 endp

```

Figure 5. KdDebuggerInitialize1 rootkit function

The load procedure is quite complex: KdDebuggerInitialize1 rootkit procedure calls Ntoskrnl function "PsCreateThreadNotifyRoutine" with a pointer to the rootkit loader Notify routine (TDLLoadNotifyRoutine). This function begin execution after first kernel thread was created. TDL Notify routine then call "IoCreateDriver" to create a driver object with a loader initialization routine (referred as TDLDrvInit). TDLDriverInit can't load rootkit driver init function because Main system Disk driver isn't still operative (and it couldn't read disk to extract Main driver). To successful resolve this trouble, TDLDriverInit installs a system callback routine (referred as TDLIOCallback) with the aid of nt function "IoRegisterPlugPlayNotification" (that is executed with EventCategoryDeviceInterfaceChange parameter and GUID of the first hard-disk class drive, which is GUID\_DEVINTERFACE\_DISK as defined in "ntddstor.h"). TDLIOCallback rootkit function is the main install procedure. It first read and decrypt rootkit File Table from last sector of the hard-disk, it find main rootkit driver relative file system sector, it then read and decrypt entire driver file. After these actions, loader must relocate the "just read" main driver, indeed it can't use system functions to load it (because they are easy to intercept and requires that driver has an entry in system registry and reside in a physical file). Loader is so much complex, and has the ability to relocate the entire driver in memory. The rootkit relocate function, after has done its job, and after having regular modified driver object for the new driver, it calls real Rootkit Driver Entry Point.

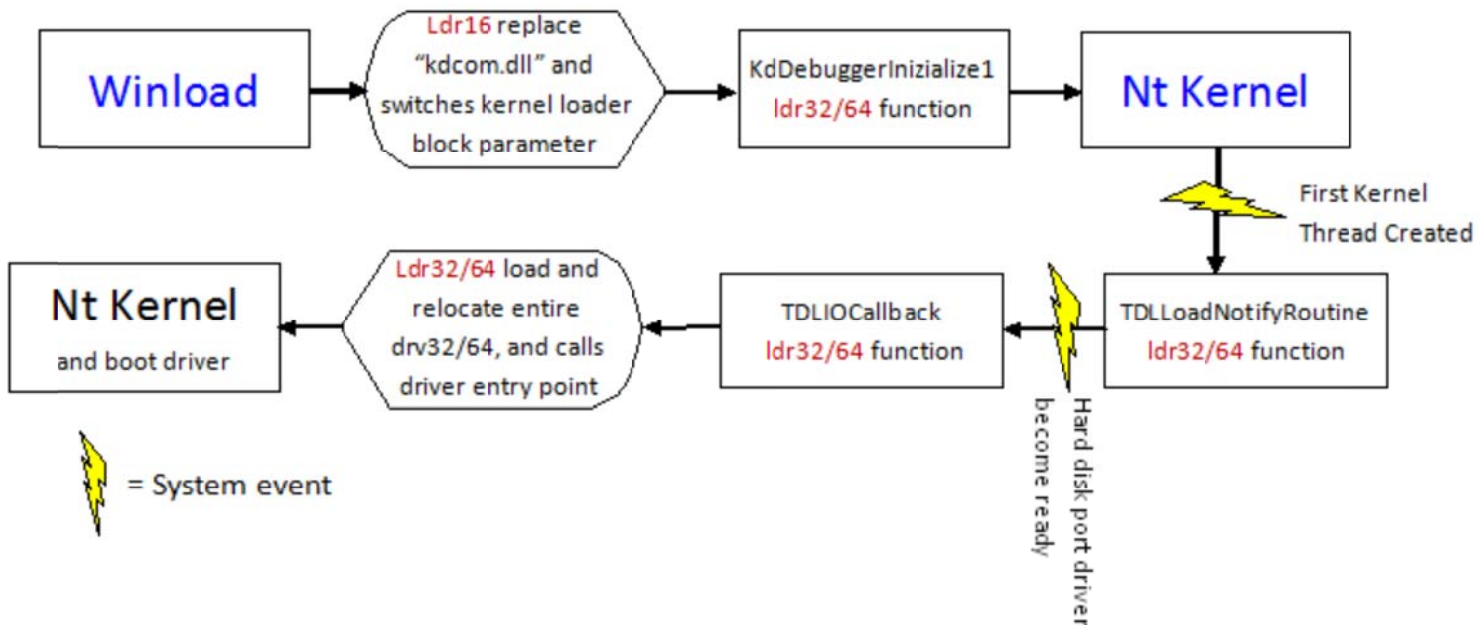


Figure 6. Main rootkit loader (ldr32/64 file) execution flow.  
Red color is for Rootkit component, blue is for original OS components

Entire infection is now ready. Main driver (drv32/64) job is to forge disk port driver and to infect entire system... but this is beyond the scope of our analysis paper...

## TDL4 - HOW TO DEBUG IT

After I have spent hours reverse engineering Ldr64 and Drv64 I found an elegant solution to this big trouble, indeed “kdcom.dll” rootkit’s replacement produce a strong side effect: it disable kernel debugger attachment support, either serial, usb, either firewire. Many tries I have done without change rootkit integrity, but this is not a valuable solution. To renable kernel debugger is necessary to inhibit “kdcom.dll” substitution. These are instructions to do it:

STEP 1. From a cleaned system, it’s necessary to read TDL4 File System from an infected one. You have to use a live cd system or a cleaned pc.

STEP 2. With a RC4 Decrypter decrypt entire File System using physical sector number as key (considering it as DWORD) and extract all TDL4 files. To do this use my own graphical TDL4 Crypter (see the last page for link).

STEP 3. Now you have to search in TDL4 “ldr16” disassembled code. To disassemble ldr16 I suggest to use the classical IDA, or Borg disassembler, a free handful open source disassembler (available here: <http://www.caesum.com/files/borg228.zip>). The Borg project is dead but if I have spare time, I would like to continue Borg project in my own, implementing support for 64 bit code... In my opinion Borg is a good project. What does the reader think?

STEP 4. Find in disassembled code the instruction:

```
26 66 81 b9 8c 00 fa 00 00 00  cmp  dword ptr es:[bx+di+8Ch], 0FAh
```

Obviously this is the 64 bit “kdcom.dll” identification instruction, 32 bit one is a couple of code line above (see Figure 4). You have to substitute this instruction with:

```
26 66 81 b9 8c 00 fa dc 00 00  cmp  dword ptr es:[bx+di+8Ch], DCFAh
```

using an Hex Editor (I suggest UltraEdit). In this way we prevent rootkit loader to identify “kdcom.dll”. This can be an issue because in this manner the rootkit will not start, but it’s the best way...

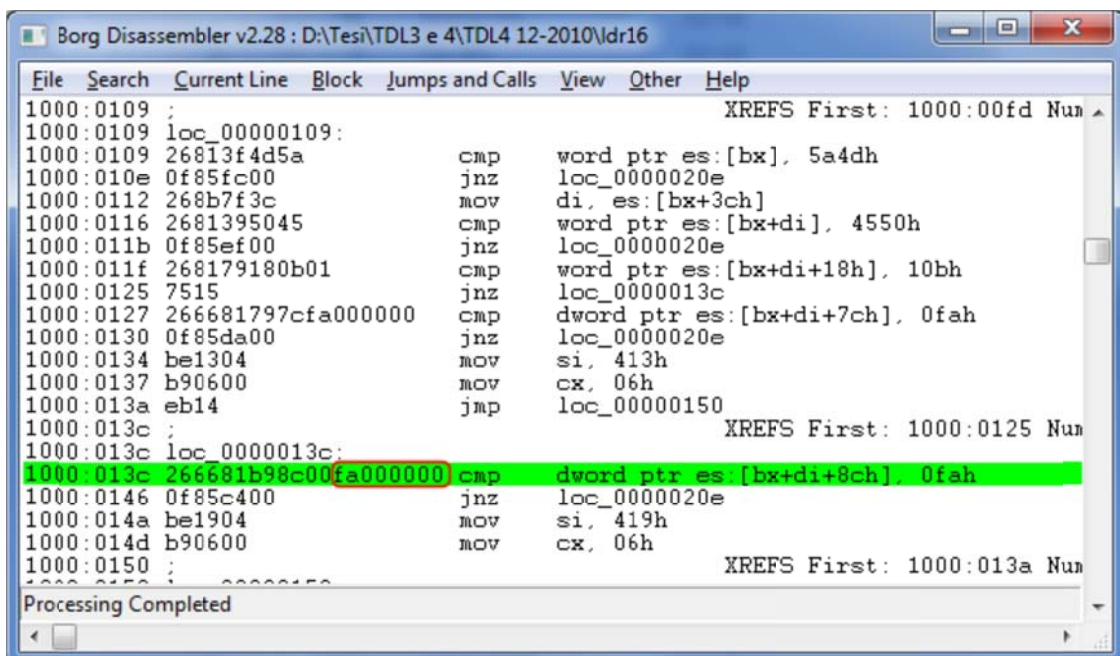


Figure 7. Byte Pattern to replace, identified with Borg

STEP 5. After have saved the file, it’s now time to re-encrypt entire Rootkit File System, and to write changes back to infected hard disk. After that, start a debugging session in infected machine. Magically the debugger now works but infection doesn’t load. To start infection is necessary to write ad hoc driver loader, linked with ldr64 rootkit file.

STEP 6. Create *ldr64 def* file with the aid of the VS2008 “dumpbin” utility (I don’t explain here how to do it because is a simple operation and because this guide is intended for an expert audience), and create a *.lib* file with the VS2008 command: “`lib /def:ldr64.def /OUT:ldr64.lib`” to linking our driver with *ldr64*. Write a own driver with this driver entry:

```
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObject, PUNICODE_STRING pRegistryPath)
{
    UNICODE_STRING drvName = {0};
    NTSTATUS ntStatus = STATUS_SUCCESS;

    DbgPrint("TDSLoader - Driver Entry Point called, be aware that this will "
            "load TDL4 infection in system!\r\n");

    // Break point for TDL4 debugging support
    DebugBreak();

    // Call rootkit ldr64 export function to launch entire infection
    KdDebuggerInitializel();

    // ... Istruzioni di inizializzazione del driver ...
    pDriverObject->DriverUnload = DriverUnload;
    return STATUS_SUCCESS;
}
```

Add *ldr64* references (having care to rename loader file to “*ldr64.dll*”), compile all and move the 2 files (driver and *ldr64*) to the infected machine. Before start debugging TDL4 it’s necessary to disable Driver Signing enforcement. Break in the debugger and enter this command:

```
kd> eb nt!g_ciEnabled 0
```

If you don’t do it, Windows refuse to load our driver, because for some reason that I didn’t exam, *SeValidateImageHeader* kernel function fail to validate *ldr64* image, even if the environment is in debug mode.

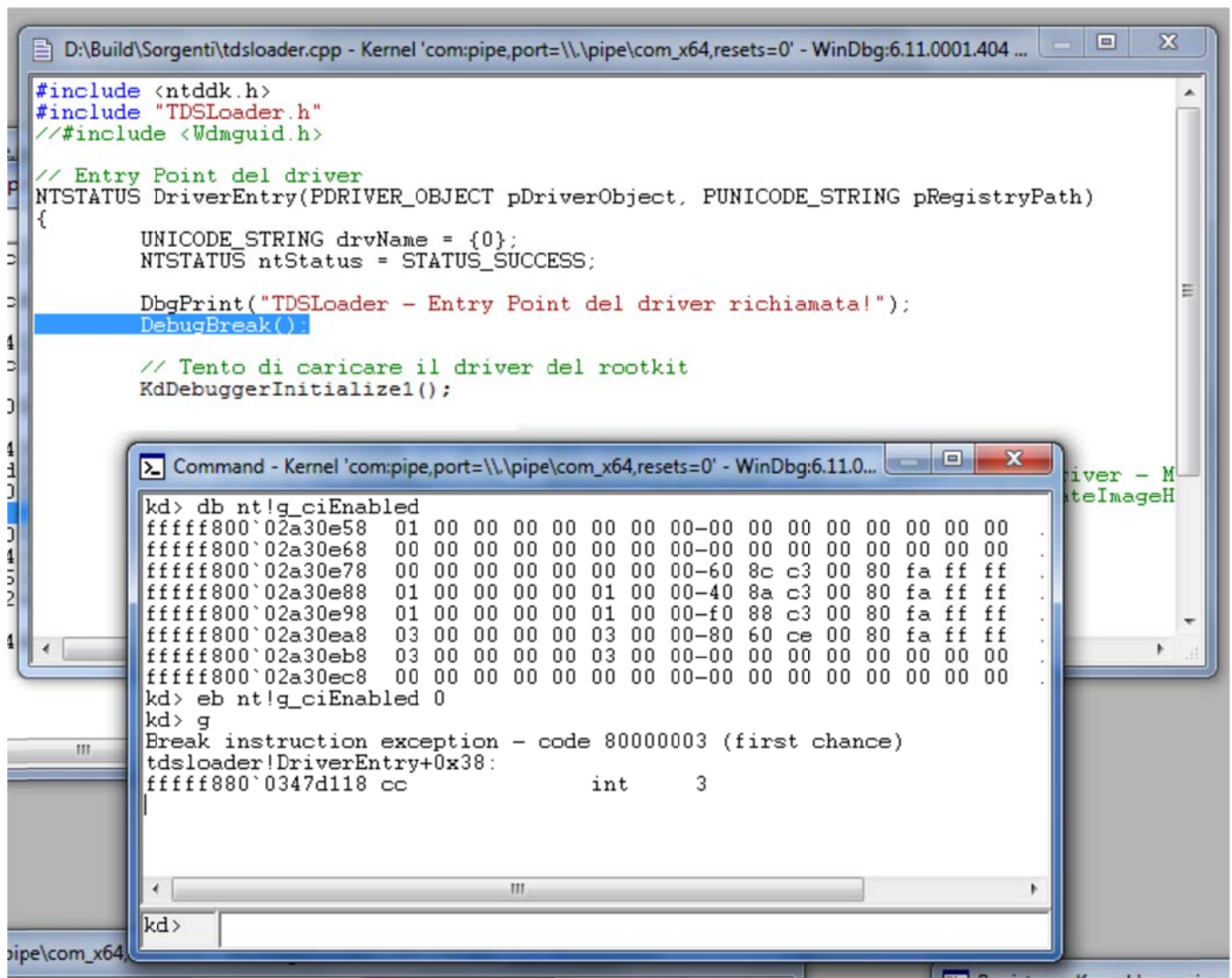


Figure 8. Our debug system is now ready to Debug entire TDL4, if the you step into KdDebuggerInitialize1 kernel function you will see TDL4 loader code

STEP 7. Install and start our driver. We are done! Debugger magically break at the Break point installed in the driver source code. If the reader steps into *KdDebuggerInitialize1* tdl function he will see all the TDL4 loader doing “dirty things”... **Well done!**

TIPS: Furthermore I suggest the reader to try insert “0xcc” int 3 function opcode around ldr16 file, having care to don’t touch early filtered hook code, but only code after the identify of bcd entry. If you enable bootmgr and winload boot debugger you will see also ldr16 code doing other “dirty things”...

## CONCLUSION

In this paper we have seen how to debug the damn TDL4 infection. I don’t covered main rootkit driver because there were other good researchers that have covered it. By the way with all these information in mind is less complex to write a custom TDL4 cleaner (like one that I wrote). With these skills I elaborate on my own, I was able to exam almost all entire TDL4 infection, except its NDIS communication protocol (I know this is a big and complex part), because I am completely ignorant about NDIS 6 programming interfaces. If someone can point out me some interesting reading on NDIS6 I will appreciate very much.

## ACKNOWLEDGEMENTS

First of all I would like to say a big thanks to Marco Giuliani (from PrevX), that shows me a lot of interesting tips about TDL3/4 and many others.

Also a thanks to all the staff of *kernelmode.info*, that provide us a lot of fresh payloads and is a big place to exchange information.

... and obviously a great thanks to TDL4 authors, that entertain me in this analysis... I'd like to say them that they made the great powerful rootkit that I ever see...

## WHO AM I

I am Andrea Allievi, nickname AaLl86, an Italian just graduated security researcher. You can contact me at [aall86@altervista.org](mailto:aall86@altervista.org). If the reader found some language mistakes please contact me, indeed English, as stated earlier, is not my native language, and I am secure that this paper is full of grammar errors.

The reader can found all TDL4 related material in this link:

<http://www.aall86.altervista.org/TDLRootkit/> (the site will become ready between 2 or 3 days, because now I haven't already post all the related material)

## REVISION HISTORY

09/01/2011 - First Release

Last revision: 08/01/2011

Copyright© 2011 by Andrea Allievi